NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE DESIGN OF A REAL TIME
OPERATING SYSTEM FOR A FAULT TOLERANT
MICROCOMPUTER

by

Robert J. Voigt

December 1986

Thesis Advisor: Larry W. Abbott

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) 62	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
THE DESIGN OF A REAL TIME OPERATING SYSTEM FOR A FAULT TOLERANT MICROCOMPUTER

12 PERSONAL AUTHOR(S)
Voigt, Robert J.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM _____ TO _____	1986 December	103

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Operating System, Real Time, Fault Tolerant, Kernel, Priority Queue

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The design and implementation of a real time operating system kernel for a fault tolerant microcomputer is presented. The operating system is designed for a real time imbedded system. The particular design is for a Motorola MC68000 microprocessor, however, the majority of the operating system is implemented using the C programming language for portability to other microprocessors. The C source for the kernel is presented. The source code is modular so that it may be used in part or as a whole operating system kernel. A heap implementation of a priority ready queue is used for task management. Performance measurements are included for parts of the ready queue.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Larry W. Abbott	22b TELEPHONE (Include Area Code) 408-646-2379 22c OFFICE SYMBOL 62At

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

Approved for public release; distribution is unlimited.

The Design of a Real Time Operating System for a Fault Tolerant Microcomputer

by

Robert J. Voigt
Lieutenant, United States Navy
B.S., United States Naval Academy, 1979

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1986

ABSTRACT

The design and implementation of a real time operating system kernel for a fault tolerant microcomputer is presented. The operating system is designed for a real time imbedded system. The particular design is for a Motorola MC68000 microprocessor, however, the majority of the operating system is implemented using the C programming language for portability to other microprocessors. The C source for the kernel is presented. The source code is modular so that it may be used in part or as a whole operating system kernel. A heap implementation of a priority ready queue is used for task management. Performance measurements are included for parts of the ready queue.

THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

The topic of this thesis is the design of a kernel for a real time operating system for a fault tolerant microcomputer system. The kernel of the operating system is the portion which contains some of the most intensely used code. A real time operating system, in the most general sense, is a system which responds to events in real time. Using this general description, most microcomputers and some minicomputers would fall into this category. For the purpose of this thesis, a real time system is one which does not require human intervention, other than to supply start-up power, to operate and has an operating system able to operate within fairly severe time constraints. The operating system must be able perform its task scheduling on a regular clocked interval, must be able to respond to outside stimulus, and must be capable of dealing with these functions in real time. The operating system should be capable of fully supporting all normal supervisory functions.

A second major feature of this operating system is that it is designed for a fault tolerant environment. A fault tolerant system is able to continue to provide critical functions after the occurrence of a fault. [Ref. 1 : p. 6] To reduce or eliminate generic software faults, each module is exhaustively tested. Several methods are available for protecting against generic software faults, including recovery blocks and, more controversially, N version programming, but exhaustive testing has been the only method to

prove effective to date. Tests are also run after the software has been integrated into the system to show that the data flow between modules is correct. [Ref 2 : p. 78]

B. MOTIVATION FOR FAULT TOLERANT SYSTEMS

The demand for fault tolerant computers is not new. The fact that the cost of microprocessors has dropped dramatically in the past few years has lead to new avenues of achieving fault tolerance in computing systems. The computing power of today's thirty two bit microprocessors, coupled with floating point co-processors, is quickly surpassing the computational power of earlier computers many times the compact size of these new micro-computer systems. Hardware fault tolerance can now be achieved inexpensivley through redundancy without sacrificing the speed and computing power of a larger system. Microprocessors also consume substantially less power which in turn leads to a solution for many of the heat problems found in older computing systems.

There is a wide range of possible applications for a compact, low power, inexpensive fault tolerant computing system. One obvious example is the use of computers in any environment where the processor will be isolated from human intervention for long periods of time as in just about any space related application. Another use for fault tolerant, compact computers is in digital fly-by-wire advanced flight control systems. Any situation where the capability to recover from a non-catastrophic fault is desired is a possible application for a fault tolerant computer.

The operating system in this thesis is designed for the fault tolerant, real time environment discussed above. An operating system designed for the real time non-interactive computing system has many advantages over the real time interactive system. One major advantage is the elimination of a major source of error, human intervention. The operating system is a static process system in that processes are not created or disposed of dynamically. All the processes the operating system will ever need have already been created and tested before the computer system is expected to function at full capacity.

The non-interactive real time operating system also has time constraints not normally associated with an interactive operating system. The operating system is an interrupt driven system which runs in a series of time slices called minor cycles. The minor cycle, approximately twenty milliseconds long for a flight control application, is the smallest period of time in which the processor must accomplish basic program functions. For example, in the case of a computer system running a flight control application, the minor cycle is determined by the frequency requirements of the flight control laws needed to fly the aircraft.

The microprocessor is interrupted every twenty milliseconds to signal the beginning of a new minor cycle and to tell the operating system that it is time to schedule a new batch of jobs to be completed during that minor cycle. Those jobs that are not completed during the minor cycle that they were scheduled in are put back in a ready queue to wait for execution or completion if a job was interrupted during its execution. One large process may be scheduled over several minor cycles, forming a major cycle.

It may be important to have a job done during a particular minor cycle or not at all. If a job is time critical and cannot be completed during the minor cycle it was scheduled in, it is removed from the ready queue without execution. In order to decide what jobs are to be done and in what order, a priority is assigned to each job. The operating system designed in this thesis uses static priority assignment. The priorities never change. In a real time system where all the possible tasks to be accomplished are known in advance, a static priority assignment is the most efficient means of dealing with prioritized jobs.

The operating system designed to operate in an autonomous environment requires fault tolerance to continue running without outside help. The operating system may run for several thousand or several million minor cycles before the computer host is shut down. During the time the computer system is running, the operating system is completely self sufficient. The operating system can schedule the necessary tasks and keep track of resources without external intervention except for incoming data. In addition, the operating system must be able to respond to external data inputs and requests in real time. The operating system in this thesis is designed to operate within the fault tolerant environment.

C. CHOICE OF HARDWARE

An objective of this thesis is the implementation of the operating system on a commercially available microprocessor. The two families of microprocessors that were considered for the implementation of the

operating system were the National Semiconductor NS32000 series and the Motorola MC68000 series microprocessors.

The microprocessor that was originally chosen for the project was the National Semiconductor NS32016 16 bit microprocessor. The NS32016 microprocessor was chosen primarily due to the simplicity and elegance of the processor design and the direct support of a floating point co-processor, National Semiconductor's NS32081. However, second sourcing agreements have have not been productive and the NS32000 chip set has failed to achieve a market share that would guarantee that it will be supported in the future. While the lack of attention for the NS32000 family may make the costs of using the NS32016 attractive, the lack of interest also means a lack of support in the critical areas of peripheral chip manufacturing by second party vendors and the lack of low power Complementary Metal Oxide Silicon (CMOS) chip sets being released and supported. The decision was made to use the more popular Motorola MC68000 microprocessor.

The MC68000 has proven to be a reliable microprocessor for use in many of today's microcomputers. The support for the MC68000 has helped to decrease its cost while time in service continues to prove that the MC68000 is a tried and true microprocessor. The MC68000 is readily available in CMOS but it does not have the direct support of a floating point unit. Even with the drawback of not having the floating point unit, the MC68000 appears to be the best choice for a mix of efficiency and effectiveness. The overall goal is to link several MC68000 microprocessor systems together and, through redundancy, achieve ultra-reliability in the overall system.

D. OUTLINE OF THE DESIGN APPROACH

Chapter II of this thesis is used to explain the concepts used in the design of the operating system and to provide a general overview of the main modules which make up the kernel of the operating system.

Chapter III details the implementation of the operating system using a Pascal-like pseudocode as a program design language. Each module is discussed in more detail.

Chapter IV describes the performance measurements obtained in the implementation of the operating system on a microcomputer. Each module is compared in a relative manner, realizing that the hardware used to test the system is not ideal.

Chapter V presents the conclusions and recommendations for further research in this area.

II. CONCEPTS

A. BACKGROUND

An operating system of some form or another is required for any computer system. Because most fault tolerant computer designs have been custom designs with little or no general purpose use and limited market share, no ad hoc or sanctioned standard for operating systems has surfaced. The operating system designed for this project will be of a more portable nature such that the operating system can be used in other systems. To achieve the portability goal, the "C" programming language, which is regarded as portable, was chosen for a majority of the operating system code.

The complexity of the computer hardware need not be reflected in the operating system. In the case of a fault tolerant design, simplicity is desired in both hardware and software. An objective of this thesis is to achieve a certain amount of robustness in the operating system design without years of test and evaluation. Another objective is to maintain a degree of simplicity in a fully functional operating system. To meet all the design objectives there may be some trade-offs in performance parameters, such as speed and efficiency. However, in order to maintain real time responsiveness, there is little room for compromise in speed. The ability to be diverse and flexible is also desired. The sum of all these qualities: speed, efficiency, robustness , diversity, and simplicity should create a satisfactory overall performance in the operating system.

Central to the concept of a real-time operating system is the ability of the computer system to respond to the randomness of the external world. To respond to these random external inputs the system needs to be able to remove some of the uncertainty which arises from these independent events and to stabilize the situation so that the process can be performed in an orderly fashion. The method by which random externals are stabilized and organized is to prioritize the events and to schedule and process these events with respect to each events' pre-assigned priority. Because a single CPU is only capable of processing one function at a time, there must be a great deal of flexibility in the system to deal with random events in addition to the regularly scheduled tasks.

B. OVERVIEW

The operating system in a real time environment is the manager of all hardware and software functions. The operating system can be thought of as multiple levels of distinct functional layers. The hardware itself forms the base layer. The kernel is the only level that has a direct interface with the hardware and has the highest software priority. The supervisor level is the level which usually supports the user level by making requests of the kernel and providing some supervisory services to the user level. The user processes form the user level and are the lowest priority processes. Because each level represents a higher priority, a hierarchy is formed. Such hierarchal designs have proven easier to debug, modify and to prove correct. [Ref. 3 : pp. 341-346] The hierarchy of a typical operating system is illustrated in Figure 2.1.

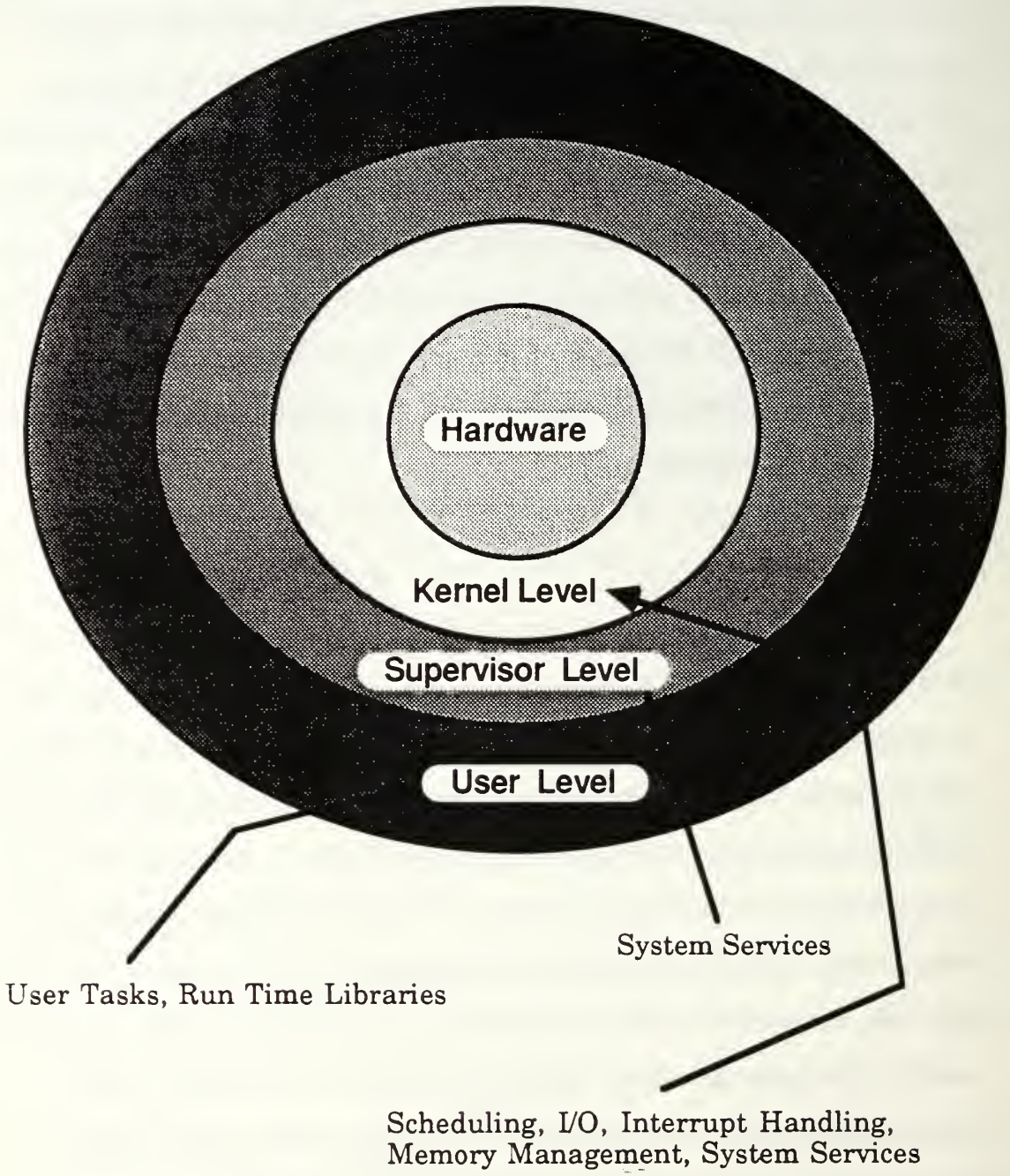


Figure 2.1 Operating System Hierarchy

This thesis is concerned mainly with the kernel level of the operating system. The user level in a real time imbedded system, such as the system that this operating system is designed for, rarely has a need for user level services as they are defined in real time interactive systems. The user processes in an imbedded system can be thought of as system level processes and thus help to simplify the design by giving all processes direct access to the kernel.

The kernel is responsible for the following functions in a real-time operating system. [Ref. 4 : p. 65]

* Interrupt handling
* Task scheduling
* Task switching
* Task suspension and resumption
* Support of Input/Output activities
* Support of memory management functions
* Support of a file system

The operating system designed in this thesis has fewer capabilities than those described above. There is no need for the support of a file system. There is no need for the support of memory management functions since each processor has its own memory that is shared globally between the operating system's processes with no dynamic memory allocations. The major functions performed by the operating system developed here are the task scheduling, suspension, resumption, switching and interrupt

handling. Interrupt handling is the most important because the operating system is an interrupt driven system.

A general state diagram listing the major functions of the operating system is shown in Figure 2.2. Each state or module will be discussed in more detail. As shown in the diagram, each state is a separate software module written primarily in the "C" programming language. The modularity of the operating system enhances its portability to other systems either in part or as a complete operating system.

C. THE CENTRAL TASK PROCESSOR

In order to reliably deal with the variety of inputs in the real-time system described in this thesis and to maintain simplicity in the design, it was decided to use a Central Task Processor through which all jobs would be processed. Each job or routine is assigned its own "instruction" which must be decoded prior to execution. The only exceptions to this rule are catastrophic error handling routines and non-maskable interrupts. The non-maskable interrupt routines are processed without maintaining the system status because of the serious conditions under which a non-maskable interrupt can be expected to occur usually means that the system will need to be reset.

In this design, the Central Task Processor uses the unimplemented instruction of the Motorola MC68000 microprocessor known as the "A-Line" trap. It is called the A-Line trap because the instruction, which is a one word value, always begins with a hexidecimal "A" in the first four bits. By assigning each task a different A-Line value in the last 12 bits of the A-Line

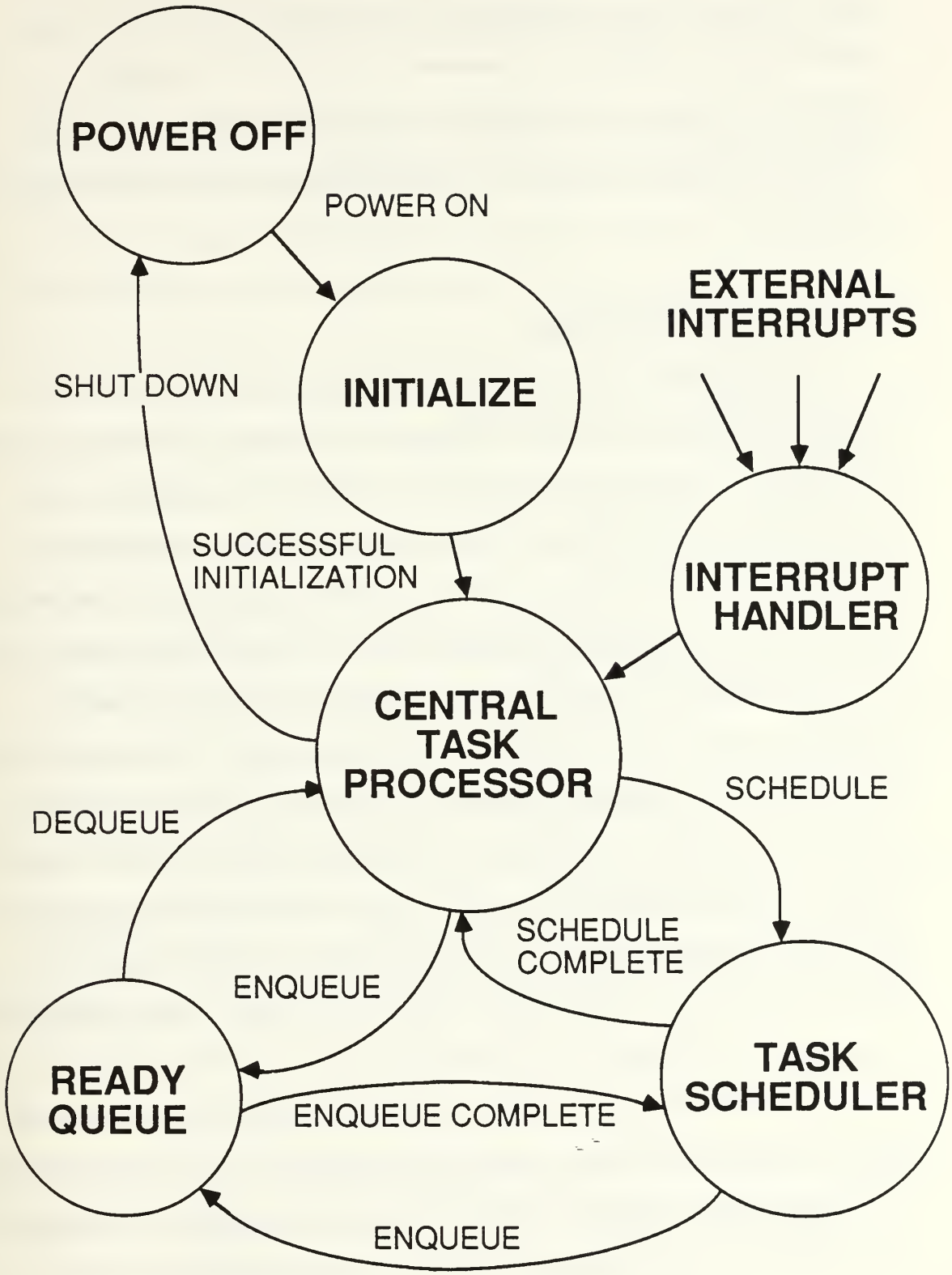


Figure 2.2 Operating System State Diagram

word, every process is defined by a one word value. This idea could be implemented on other microprocessors by the use of a trap style instruction, however, the flexibility and the large number of tasks that could be described in a single word made this an obvious advantage for the MC68000.

D. THE TASK SCHEDULER

The Task Scheduler is responsible for the scheduling of tasks at the beginning of each minor cycle. The operating system is responsible for the execution of a sequence of minor cycles, which are initiated by a timer interrupt generated within the computer system. The timer interrupt causes the operating system to suspend the task that is being executed and to start the execution of a new minor cycle. The first thing that happens in a new minor cycle is to save the state of the suspended job and begin scheduling a new set of jobs based on the frame number of the new minor cycle. The frame number is maintained by the frame counter which is a thirty two bit quantity which has been initialized to zero during the start-up phase. It is the Task Scheduler's job to insure that the frame counter for the system is incremented. For aircraft controls, the typical time for a minor cycle is ten to twenty milliseconds. For example, twenty milliseconds was used in the Digital Fly-By-Wire Flight Control Validation Experience for the F-8 aircraft using an AP-101 computer. This twenty millisecond time for a minor cycle can be varied for experimental purposes and should be adjusted to be a suitable amount of time for the program functions to be accomplished. Some functions do not have to be completed in a single

minor cycle but require several minor cycles forming a major cycle. [Ref. 1 : pp. 95-96] Figure 2.3 illustrates the sequence of execution of a minor cycle.

A desired feature of the operating system is the ability to distribute the tasks evenly among the minor cycles so as to make better use of the processor throughput. The jobs are scheduled according to their sample time requirements, and normally must be completed in the cycle they are originally scheduled in. Certain jobs are scheduled every cycle while some may be scheduled every other or every fourth cycle. The jobs which must be completed during the cycle in which they are scheduled will be marked so that if a job does not complete execution during the minor cycle in which it was originally scheduled, the job will be purged from the Ready Queue at the end of the current minor cycle.

There will be a requirement to indicate which jobs have been purged from the Ready Queue, especially in the final determination of the length of the minor cycle. The possibility exists that a job may not be executed until the frame counter has turned over and is on the same frame number again, however, with a thirty two bit frame counter, this job would prove to be delayed 4,294,967,296 times. Assuming a twenty millisecond minor cycle, a job would have to be delayed 994.2 days or 2.724 years. This is an acceptable risk.

E. THE READY QUEUE

The heart of the operating system kernel lies in the two most important functions of the Ready Queue and the Task Scheduler. The major

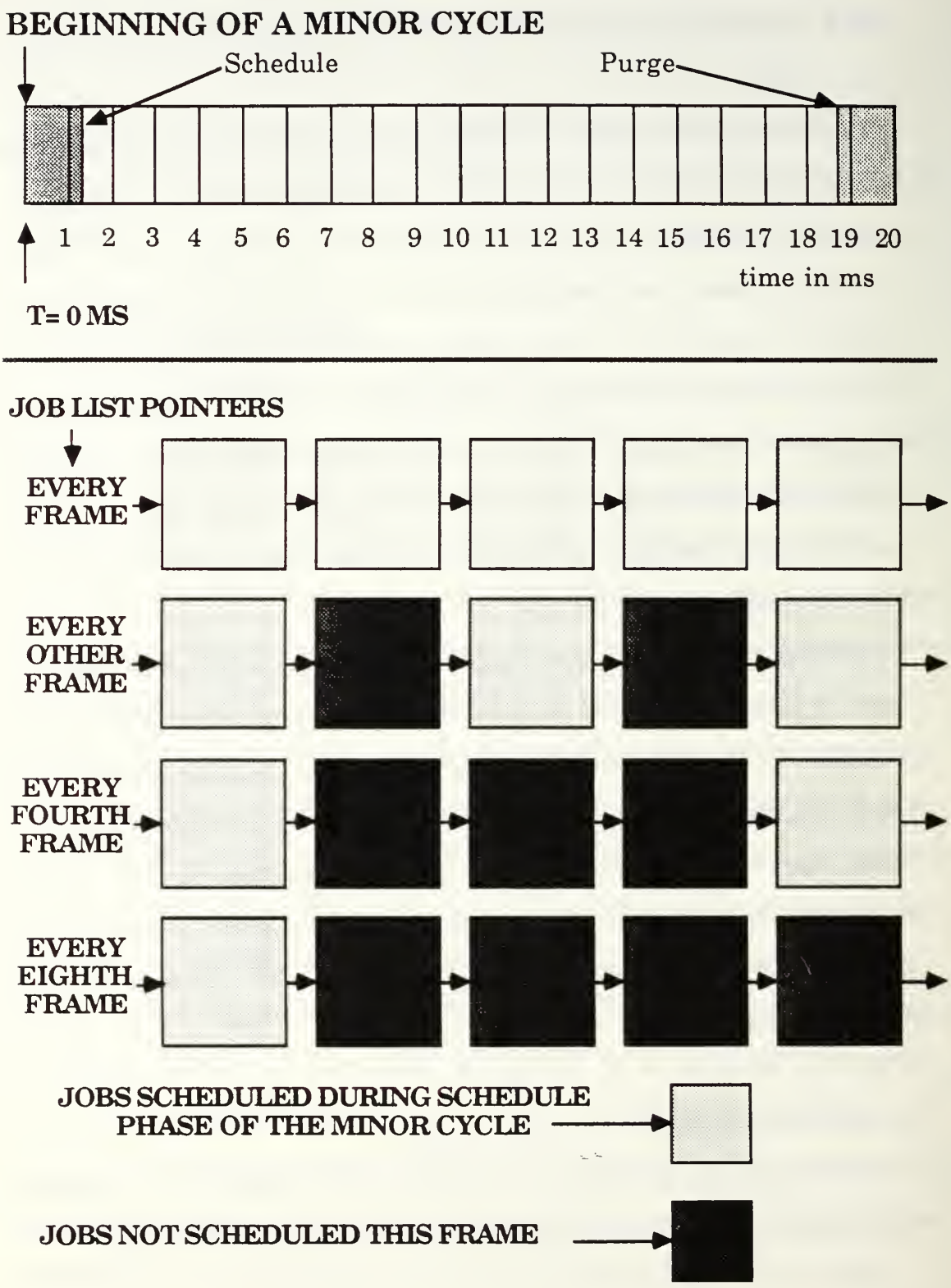


Figure 2.3 Minor Cycle Timing Sequence

requirements for these two critical areas are speed and efficiency. A major percentage of all the code executed by the system lies in the maintenance of the Ready Queue. The Ready Queue is described by a binary tree data structure known as a heap. A heap is implemented using a sequential representation based on an array. The array elements form a complete binary tree such that each parent is less than either of its children. In order to achieve a maximum speed and efficiency the Ready Queue was designed as a heap implementation of a priority queue.

There are several means of implementing a priority queue. The major reason for choosing a heap implementation is that it is the most efficient method for a non-trivial number of elements in the queue for the two critical queue operations of enqueueing and dequeueing. Figure 2.4 summarizes three types of queues and the approximate orders of magnitude for a single insertion and deletion in each type of queue.

	List		
	Array	**Linked**	**Heap**
enqueue	O(n)	O(n)	$O(\log_2 n)$
dequeue	O(1)	O(1)	$O(\log_2 n)$

Figure 2.4 Performance Estimates of Priority Queues [Ref. 5 : p. 108]

Figure 2.5 illustrates graphically the relative orders of magnitude differences between the List and Heap priority queues. The lists are more efficient at dequeueing but the heap is considerably better overall as the number of elements in the queue, n, becomes large.

The Ready Queue is also capable of having a job of the same type scheduled more than once and will treat these on a first in first out basis. Two jobs will never actually be identical since a serial number will be assigned to the job as it enters the queue.

For purging operations, the heap implementation is approximately equivalent to the list implementation since the queue is represented by an array of pointers. The heap implementation actually has less housekeeping for keeping track of free space in the queue than the list implementation. Overall, the heap implementation appears superior.

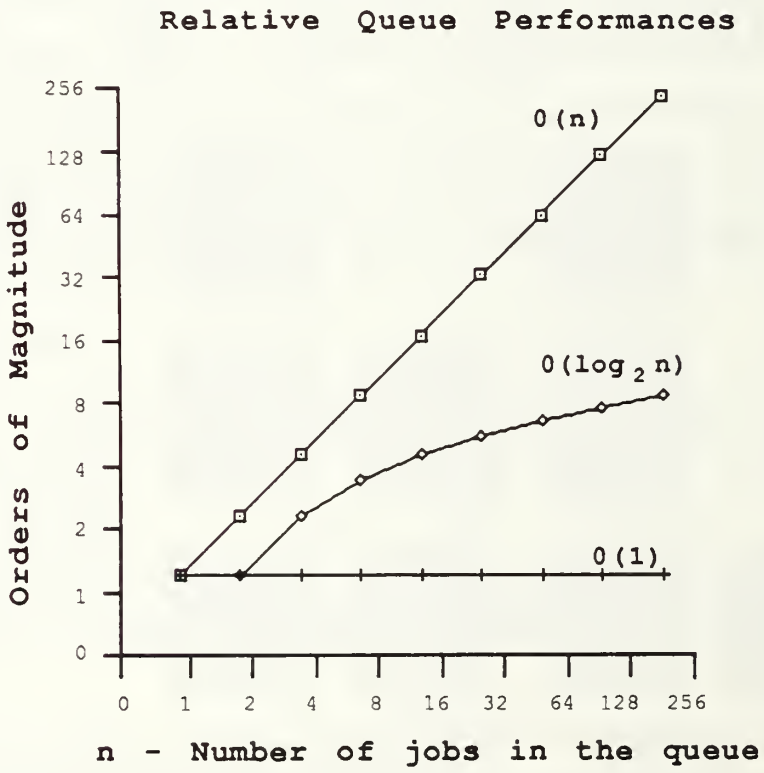


Figure 2.5 Relative Orders of Magnitude

F. THE INTERRUPT HANDLER

There are several interrupt handlers that are used in the operation of an operating system. The one which will be discussed here is the Cycle Interrupt Handler. The Cycle Interrupt Handler is the routine which is called after the microprocessor has been interrupted for the beginning of a new frame or minor cycle. Two important functions that must be performed at the beginning of a new frame are to schedule all the jobs in the queue for the new minor cycle and to purge any jobs which remain in queue but are no longer valid. A job is no longer valid when it is marked as a job to be completed within the current minor cycle but its frame number does not match the current minor cycle's frame number.

The Cycle Interrupt Handler is responsible for saving the status of the job that was active at the time of the interrupt and to call the scheduler and purge functions. After completing the scheduler and purge calls, the Cycle Interrupt Handler returns the control of the operating system back to the Central Task Processor. The Central Task Processor then will perform its function and activate the job with the highest priority.

G. SUMMARY

The basic concepts discussed for each of the main modules of the operating system have been reviewed in a very general manner. In the next chapter the implementations will be discussed.

III. IMPLEMENTATION OF THE OPERATING SYSTEM

A. INTRODUCTION

In this chapter the details of the implementation will be discussed. The main modules, the Central Task Processor, the Cycle Interrupt Handler, the Ready Queue and the Task Scheduler will be illustrated in a Pascal like pseudocode and flow diagrams. The actual "C" code and assembly language code is found in Appendix A.

In order to avoid confusion and to set up a basis for the rest of the chapter, some definitions are provided in Table 3.1. The definitions used in this thesis may not strictly adhere to definitions used in the development of other operating system designs but the difference occurs only when a standard definition is not widely accepted as accurate.

TABLE 3.1 TERMS AND DEFINITIONS

Term	Definition
A-Line	An A-Line is a one word (16 bit) value whose first 4 bits are a hexidecimal "A".
A-line Trap	A feature of the MC68000 microprocessor which allows an extension to the normal instruction set through the use of A-Lines.
C	A general purpose, relatively "low level" programming langauge.
CTP	The Central Task Processor through which all tasks are processed.

TABLE 3.1 -- Continued

Term	Definition
Dequeue	Removing an item from the queue.
Dispatch	The process of decoding a job number and starting execution the job.
Enqueue	Adding an item into the queue.
F-Line	An F-Line is a one word (16 bit) value whose first 4 bits are a hexidecimal "F".
F-line Trap	A feature of the MC68000 microprocessor which allows an extension to the normal instruction set through the use of F-Lines. This feature is reserved for the addition of the MC68881 Floating Point Coprocessor.
Failure	A condition which can give rise to a fault, usually considered permanent.
Fault	An anomaly in the performance of the system.
Fault Tolerant	A system which is able to continue to provide critical functions after the performance of a fault.
Interrupt	An external condition which causes the microprocessor to stop execution to service a request.
Job	A specific set of code which defines a process and is assigned a prioritized number. Interchangeable with task.
Kernel	The portion of the operating system for the control and management of all operations that involve processes and resources.

TABLE 3.1 -- Continued

Term	Definition
Major Cycle	An integral number of minor cycles needed to perform an entire function.
Minor Cycle	The smallest interval of time needed to perform the basic program functions
Monitor	A program which allows the monitoring and alteration of the system.
Nucleus	Another term for kernel.
Operating System	Programs which allow control of the hardware, making it usable.
Priority Queue	A hierarchal queue.
Process	A program in execution.
Purge	Removal of a no longer needed item from the queue.
Queue	A data structure which contains information about processes.
Ready Queue	A queue whose processes are ready to be executed.
Real-Time Operating System	An operating system which responds to events as they occur.
Task	A specific set of code which defines a process. Interchangeable with job.
Watch Dog Timer	A timer which interrupts the system after a set amount of time has elapsed.

In addition to designing the program structure, the design of the operating system also includes the design of data structures and "structures" as defined by Kernighan and Ritchie in Reference [6]. There is also the A-Line trap word which in itself is a type of data structure. Figure 3.1 shows the makeup of an A-Line word.

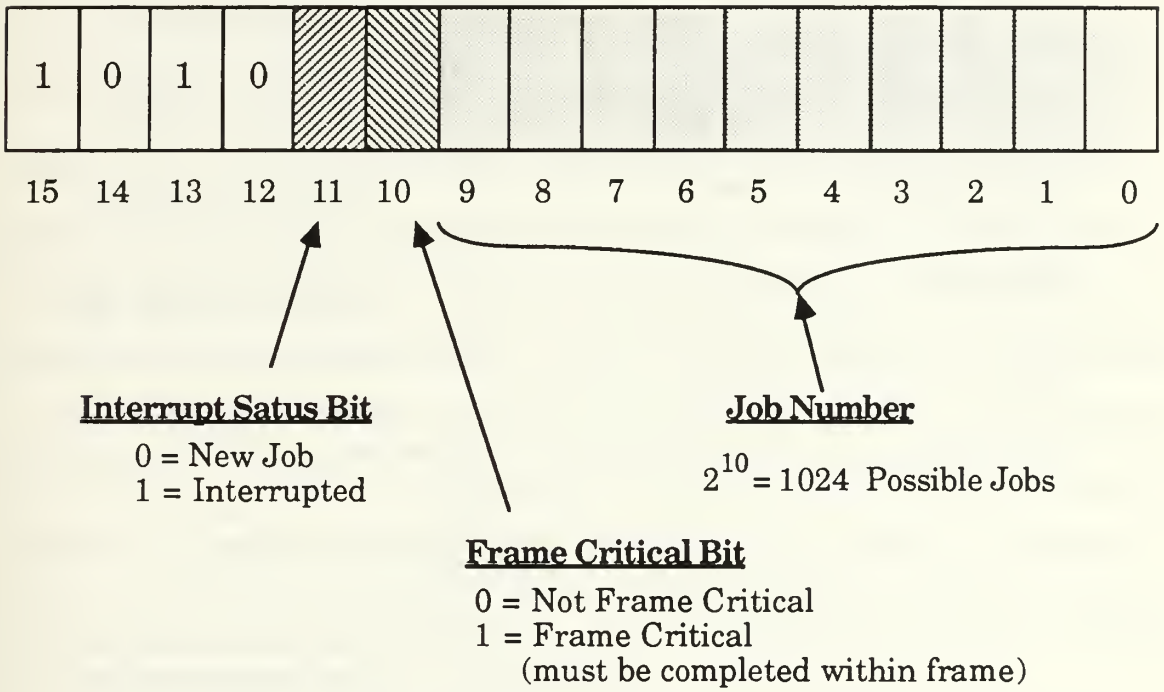


Figure 3.1 A-Line Trap Word

In this format, the last twelve bits of the word completely describe a job. The information in the A-Line coupled with a dispatch table gives the location in Read Only Memory of the job to be executed.

The Ready Queue is the only other major data structure which needs to be explained. The Ready Queue is a heap implementation of a priority queue. The queue itself is an array of pointers which point to an array of nodes which hold the data for each job in the queue. In this fashion all

memory for the queue is allocated at the time of the definition of these arrays. Figure 3.2 shows the data structures involved in the Ready Queue.

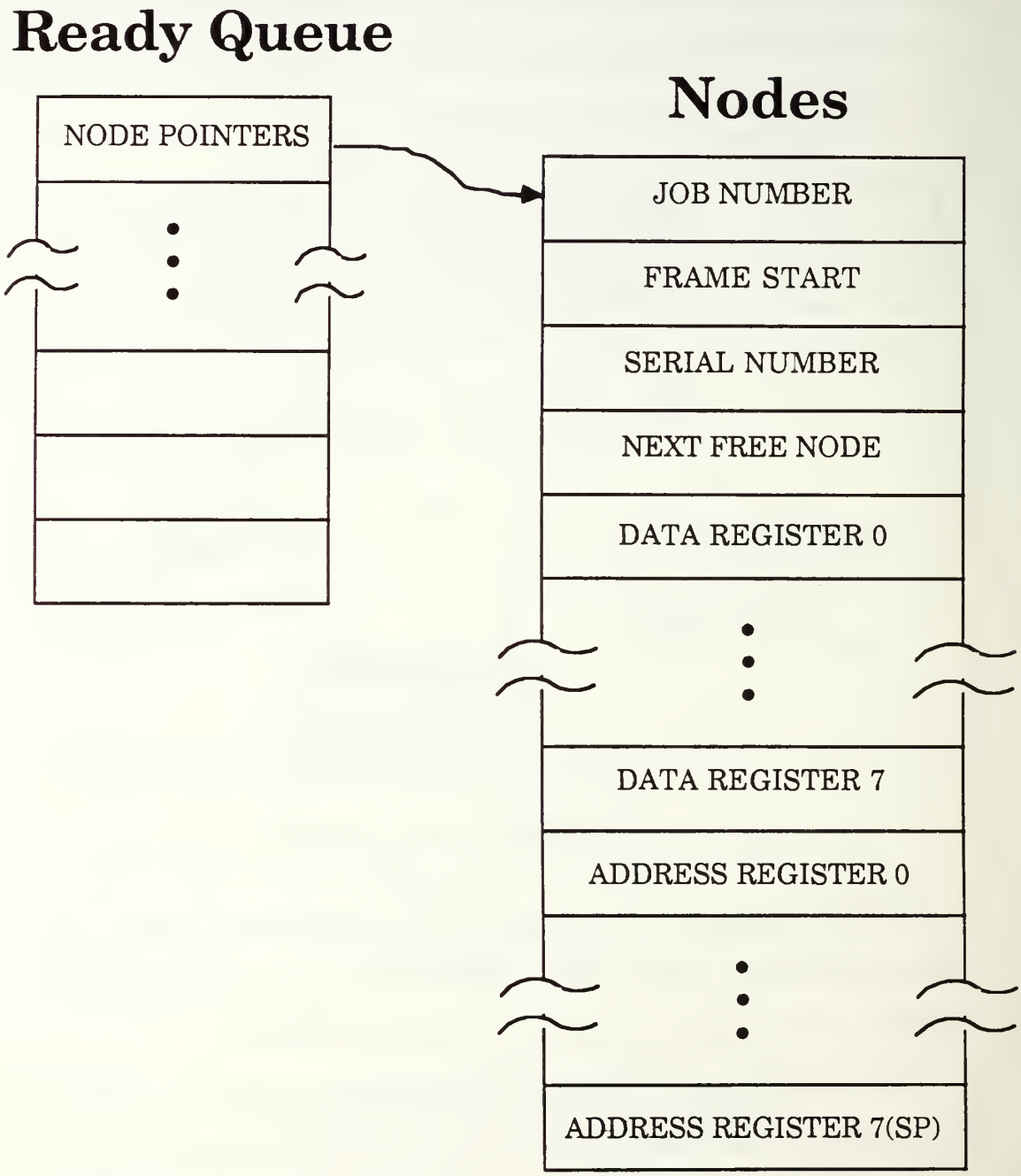


Figure 3.2 Ready Queue Data Structures

B. SYSTEM OVERVIEW

The operating system is confined to performing a limited number of services. The reason for these limitations is that the operating system is not meant to be generic or multi-purpose. This operating system is designed to perform a pre-defined number of tasks that are located in Read Only Memory (ROM) and are therefore somewhat permanent. The operating system itself is designed to be ROM resident using Random Access Memory for the dynamic components of the system such as the Ready Queue. All of the memory for the Ready Queue and the variables required by the tasks is pre-allocated during the initialization phase of the system. By not allowing any dynamic memory allocations, the possibility of a task overwriting another task's memory space is reduced. By eliminating accidental memory overwrites, a form of memory protection is added to the fault tolerant capabilities of the system. The value of the memory protection scheme is increased because a priori knowledge exists about the memory requirements of every possible task. With this increased certainty, virtually all memory overwrites can be eliminated.

Figure 3.3 shows the basic flow for the operating system. Not all of the modules shown were implemented in "C". There are some functions, which must deal with hardware, which are not practical or convenient to write in "C". Some portions of the operating system are executed so frequently that it is preferable to write these parts of code in assembly language for execution efficiency. Even the UNIX operating system kernel contains 1000 lines of assembly code along with its 10,000 lines of "C" code.[Ref. 4 : p. 484] Those modules written strictly in assembly language

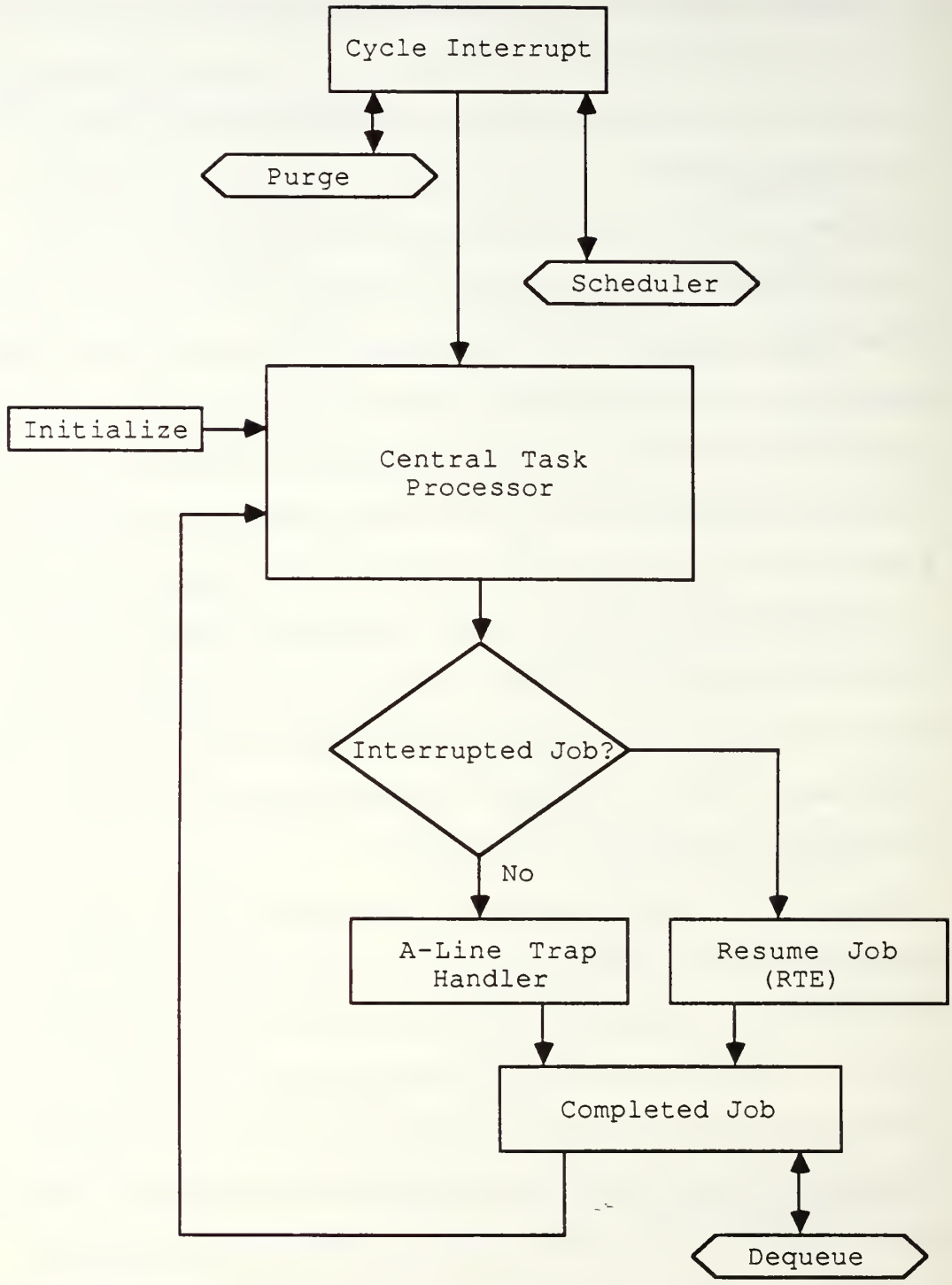


Figure 3.3 General System Flow

code were the A-Line Trap handler and the main part of the Central Task processor. For convenience, the portions of code dealing with peripheral chip initialization and support were also written in assembly language.

As shown in Figure 3.3, the heart of the operating system lies in the Central Task Processor. The Central Task Processor is made up of the A-Line Trap handler, and the Job Dispatch Table. All jobs, whether previously interrupted or not, will be processed by the Central Task Processor. While the Central Task Processor may be some of the most heavily used code, it is not particularly complex and therefore the operating system does not spend a large percentage of time executing this module.

The Ready Queue, with all of its associated maintenance, is the area where the operating system spends most of its time. In order to increase the efficiency of the operating system as a whole, close attention must be paid to the efficiency of the Ready Queue.

The interrupt handler, denoted as the Cycle Interrupt module in Figure 3.3, is a small portion of code which is also written in assembly language code. The interrupt handler is an additional area of necessary overhead which must be optimized. The Cycle Interrupt module calls the Task Scheduler and the Purge routines which in turn call routines that add to and delete from the Ready Queue, respectively.

C. THE CENTRAL TASK PROCESSOR

All the jobs are processed through the Central Task Processor to increase fault tolerance. The use of a centralized area of the operating system to monitor the start of any new job or the completion of any

interrupted job was intended to help the operating system monitor the system and thereby increase its resistance to faults. The Central Task Processor includes the A-Line Trap handler and the Dispatch Table for execution of new jobs. The Central Task Processor is also responsible for the continuation of old jobs that have been previously interrupted.

Figure 3.4 shows the main part of the Central Task Processor which was implemented in assembly language code to facilitate the use of the A-Line Trap and the direct manipulation of registers. The first task is the retrieval of the job number. Because the Ready Queue is a priority queue, the next job to be executed will always be the top job in the queue. In the one word A-Line value, which represents the job number, bit eleven is the interrupt status bit. If the job has been previously started and interrupted this bit will be set.

There are two ways in which a job can be executed. First, if the job is a new job, the A-Line value will be loaded into an address location to be executed as an unimplemented instruction. Second, if the job had been started and then interrupted, the microprocessor is restored to the state it had immediately after the job was interrupted, and then a return from exception (RTE) instruction is executed.

The return from exception instruction will take the top six bytes off the stack and resume execution at the restored program counter (PC) which was stored on the stack. The old status register will also be restored. Since the stack pointer which is restored prior to the execution of the RTE is pointing to the the status of the job when it was interrupted, execution continues where the job was at the time of the interrupt.

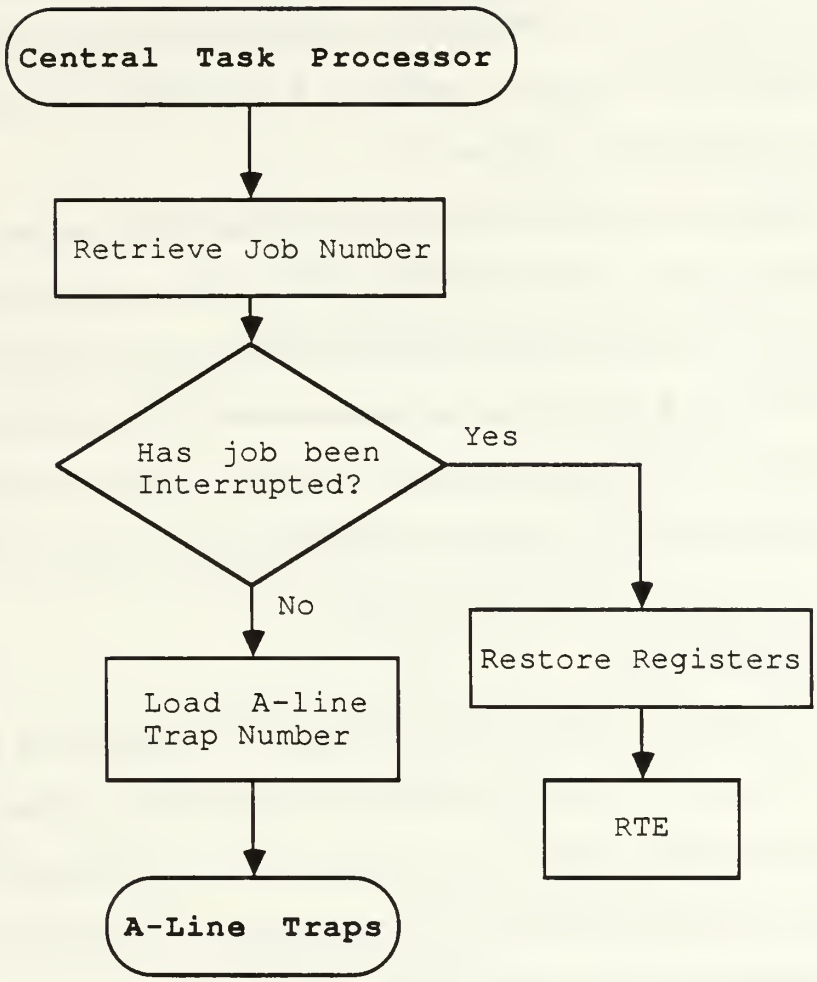


Figure 3.4 Central Task Processor

1. A-Line Traps

The routine for the execution of a new job is to load the A-Line Trap into a pre-defined address so that it will be executed as the next instruction after the load. Once the program counter encounters this A-Line the exception handler routes execution to the routine whose address is in the vector table in low memory. This is the A-Line Traps module whose flow diagram is illustrated in Figure 3.5.

The first thing done in this exception handling routine is to inhibit any other interrupts. Since the exception handling routine of the MC68000 has pushed the status register and the program counter onto the stack and there will not be a return from exception instruction executed from this routine, the stack must be adjusted. The next thing to do is fetch the job number that caused the exception in the first place. Based on that job number, the Dispatch routine is called.

2. Dispatch Table

The Dispatch routine is a "C" routine which sends the processor off to execute the job. Up until and during execution of the Dispatch routine, interrupts have been inhibited. It is the responsibility of the job being executed to lower the interrupt priority set by the A-Line Traps module. The pseudocode in Figure 3.6 illustrates the basic switch statement used to dispatch to the job.

Upon completion of any routine, whether it was interrupted or not, the routine must be removed from the Ready Queue. The responsibility for removal is in the JobDone routine. The JobDone routine calls the queue function, Dequeue, which removes the top job from the Ready Queue. Then

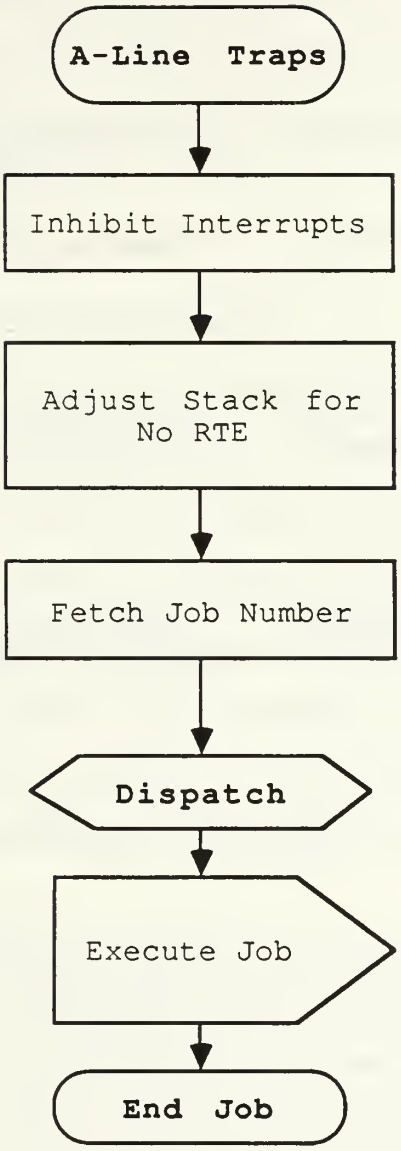


Figure 3.5 A-Line Trap Handler

JobDone adjusts the stack, because it was called as a subroutine and will not return to the caller. Control is then passed back to the Central Task Processor and The Central Task Processor then continues on with the next job.

```
procedure Dispatch              /* job dispatch table */
begin                           /* limited number of jobs version */

switch(PresentJob)              /* based on job from A-Line Trap */
begin

        case 1:
    begin
            DoJobOne;           / * handle Job 1 */
            JobDone;            /* After Job is completed */
    end;

        case 2:
    begin
            DoJobTwo;           / * handle Job 2 */
            JobDone;            /* After Job is completed */
    end;
end;
end;
```

Figure 3.6 Dispatch Routine

The decision to use A-Line traps in the operating system was made early on in the design phase. The original idea was that with each job assigned its own "instruction" a job could be very easily called from anywhere in any task's code for execution. The job could be quickly decoded and executed and control returned to the calling program with an return from exception instruction. Another consideration was that there could be a total of 1024 different jobs defined in one word values. This seemed to be an advantage over the limited user defined traps provided by the MC68000.

There are only 192 user defined traps for the MC68000 and their execution is slower. As it turns out, the A-Line trap could be replaced by a simple table of contents eliminating a great deal of unnecessary exception handling code. A table of contents could be constructed from an array of starting addresses with the array subscripts equal to the job numbers. The priority would still be encoded in the job number. This will be discussed further in the conclusions.

D. THE CYCLE INTERRUPT HANDLER

The Cycle Interrupt Handler is the routine which responds to the external signalling of a new cycle or start of a frame. Depending on the hardware configuration, the external interrupt is assigned an interrupt priority. Based on this priority, a vector table in low memory points to the routine which handles the interrupt. On the MC68000, there are seven levels of interrupts available. In the microcomputer that the operating system for this thesis was developed on, a Synertek SY6522 Versatile Interface Adapter (VIA) was used to generate a level one interrupt to signal the start of a new frame.

Figure 3.7 illustrates the flow diagram for the Cycle Interrupt Handler. The first and most important responsibility of the Cycle Interrupt Handler is to store the data and address registers in the data structure for the job that was interrupted. The data structure will be the top job in the Ready Queue. After storing the registers, the interrupt bit of the job is set so that the Central Task Processor will know that the interrupted job had started execution.

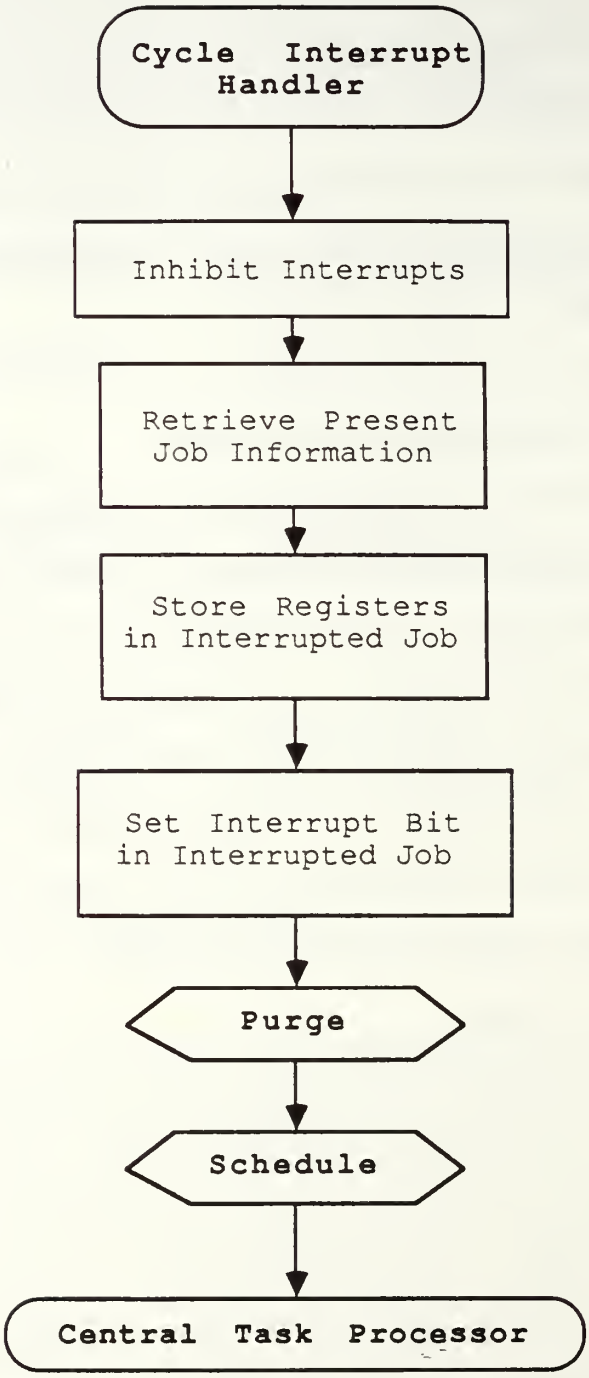


Figure 3.7 Cycle Interrupt Handler

The Purge and Task Scheduler routines are called next. Purge is a Ready Queue maintenance routine which removes the residue of jobs which had to be executed in the last frame from the Ready Queue. The Task Scheduler then enqueues all the new jobs for the new frame. Upon completion of the Purge and Task Scheduler routines the Cycle Interrupt Handler returns control to the Central Task Processor for execution of the next job.

E. THE READY QUEUE

1. Heap Data Structure

The Ready Queue is represented as a heap implementation of a priority queue. A heap is a restricted binary tree data structure. The Ready Queue is a sequential array of pointers. The requirements for a sequence to be a heap is that each parent must be less than either of its children. If r[1], r[2], ... r[n] is a sequence of elements, the sequence is a heap if r[i] < r [2i] and r[i] < r[2i+1]. Lower job numbers in the heap represent a higher priority job.

2. Enqueue Routine

The flow diagram for the Enqueue module is shown in Figure 3.8. The Enqueue routine calculates a serial number to insure that if the job is added to the queue twice in one frame it is treated on a first in-first out basis. The serial number for each job is located in an array of serial numbers indexed by job number. To make room for the serial number, the job number, which is an A-Line, is shifted four bits to the left. The job is

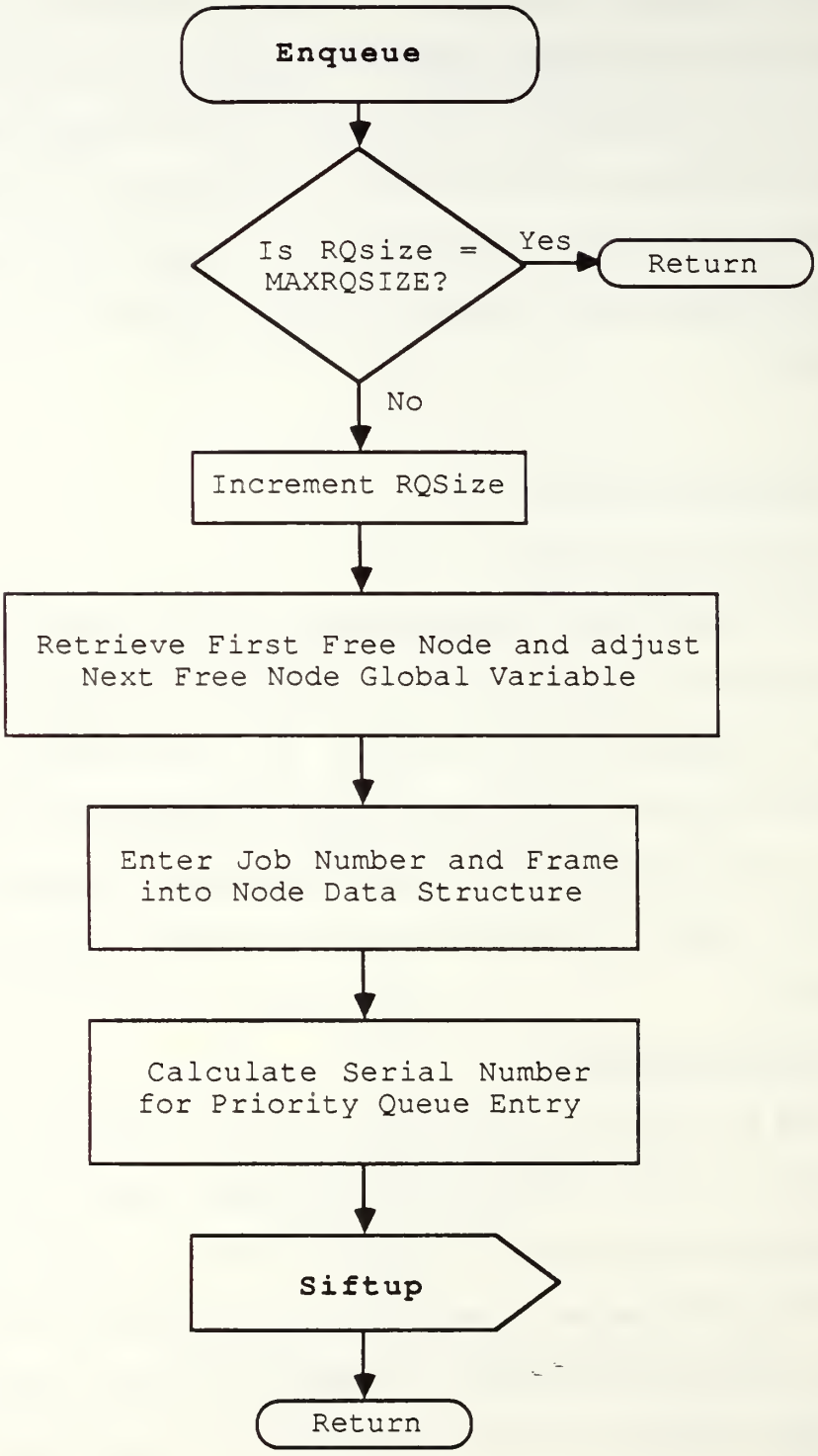


Figure 3.8 Enqueue a Job

then assigned the lowest position in the queue and the Siftup routine is called.

A simplified Enqueue routine is shown in Figure 3.9 using data structures and variables similar to those used in the actual Ready Queue. The elements are added to the queue using the Enqueue routine.

```
procedure enqueue( A-Line : jobNumber)
 begin
   RQSize = RQSize + 1;                          /* increase RQ Size  */
   RQNode[RQSize].JobNumber = jobNumber;      /* fill new node */
   RQNode[RQSize].FrameStart = FrameCounter;
   siftup(RQSize);                               /* restructure heap */
 end;
```

Figure 3.9 Enqueue - Add an Element to the Queue [Ref. 5 : pp. 243-245]

3. Siftup

The operation Siftup moves an element of the queue toward the root. Siftup is restricted to a single path from the leaf node to the root of the heap. Because of this single path, it will require at most $0(\log_2 n)$ effort to add an element to a queue of size n. Figure 3.10 shows the Siftup routine. The element which is being "sifted up" is moved from the location "position" toward the root until the heap conditions are satisfied.

Figure 3.11(a) shows a heap formed from a sequence of integers. The integers in Figure 3.11(a) were added to the queue in the following order : 015,004 023,010,036. The numbers alongside the elements in Figure 3.11(a) represent the array subscripts of the integer sequence. Figure 3.11(b) shows the heap in Figure 3.11(a) after adding the element 012 using the Enqueue and Siftup functions.

```
procedure siftup(int : position)  /* Move element towards the Root   */
 begin                            /* from position until heap is  satisfied */

 var
  int :          j,k;

   A_Line:    jobNumber;                              /*  jobNumber is Priority */
  jobNumber = RQNode[RQSize].JobNumber;  /*  retrieve the job number */
  RQNode[0] = RQNode[position];                 /*  save the present node  */
  k = position;                                        /* save the present  position */
  j = position div 2;

 while RQNode[j].JobNumber > jobNumber do
   begin
     RQNodePtr[k] = RQNodePtr[j];                /* find proper location */
     k = j;
     j = j div 2;
  end;

   RQNode[k] = RQNode[0];         /* location found move element there */
end;
```

Figure 3.10 Siftup - Move an Element Towards the Root [Ref. 5 : pp. 243-245]

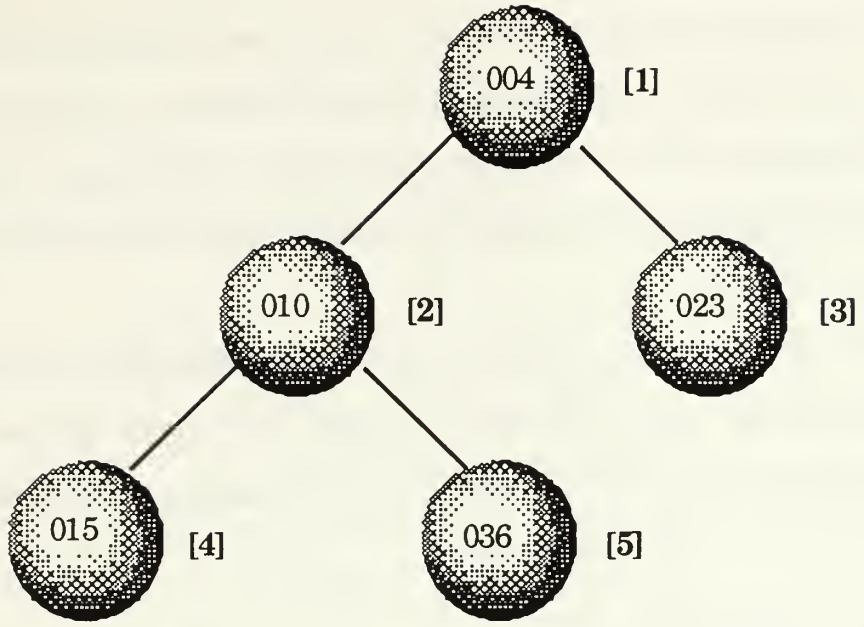


Figure 3.11(a) Heap Implemenation of a Priority Queue

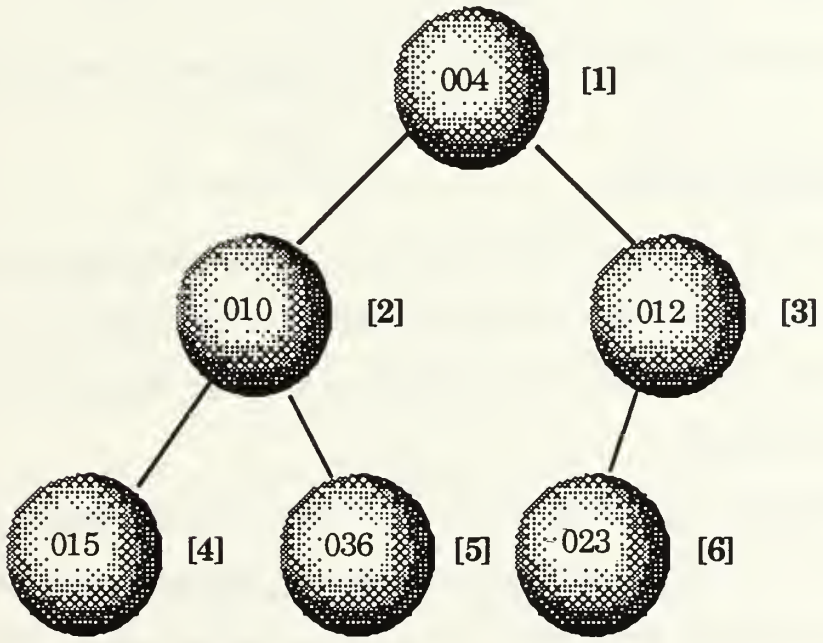


Figure 3.11(b) Heap After Element 012 is Added

4. Dequeue Routine

To remove an element from the queue, the Dequeue routine is called. Figure 3.12 shows a pseudocode routine for Dequeue. The Dequeue function is illustrated in a flow diagram in Figure 3.13. The Dequeue function is a relatively simple operation with the complex operations taking place in the Siftdown routine.

The Dequeue routine shown does not return a value, however it could return the node which was removed. Figure 3.14 shows the heap of integers from Figure 3.11(b) after calling the Dequeue module. The element 004 was removed and the heap restructured.

```
procedure dequeue
begin
      /* Put a lower priority job on top of the heap */
  RQNodePtr[1] = RQNodePtr[RQSize];
  RQSize = RQSize -1;              /* decrease RQ Size  */
  siftdown(1);                     /* restructure heap  */
end;
```

Figure 3.12 Dequeue Top Element From the Queue [Ref. 5 : pp. 243-245]

5. Siftdown Routine

The function called Siftdown moves an element from the root towards a leaf node. The process continues until the heap conditions are satisfied. This process again requires at most $0(\log_2 n)$ effort. Figure 3.15 shows the Siftdown procedure.

The operating system designed for this thesis has modified the pseudocode modules presented in the figures by using pointers to data nodes in the actual heap rather than the nodes themselves. The reason for using pointers is that it is easier to shift a single value up and down the tree

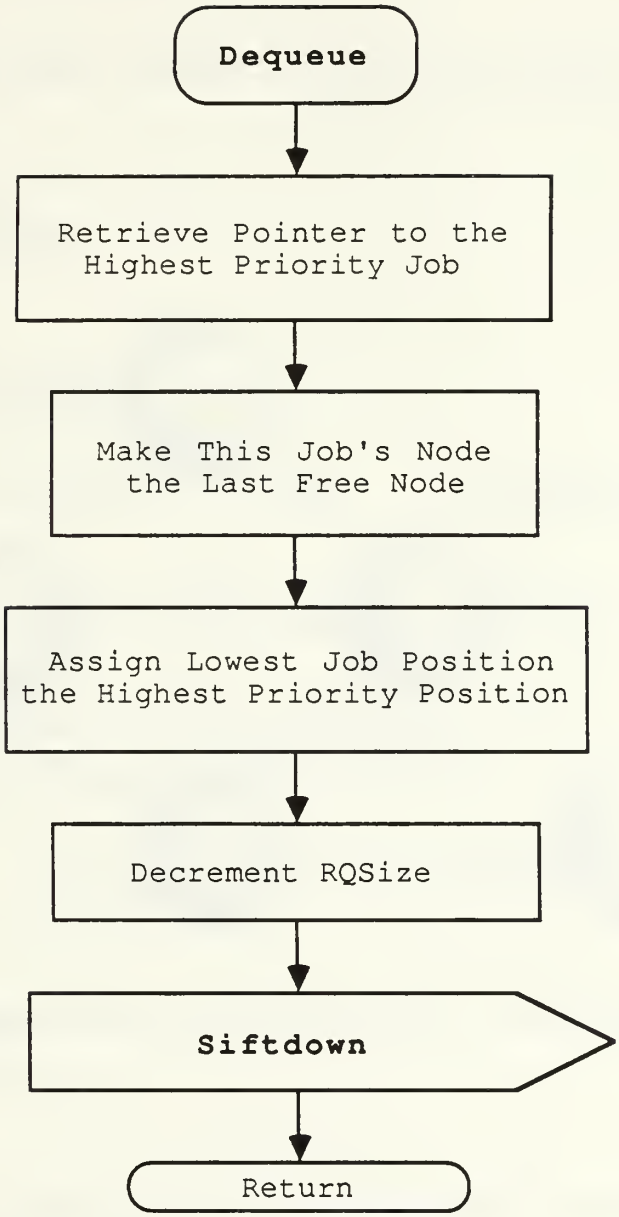


Figure 3.13 The Dequeue Module

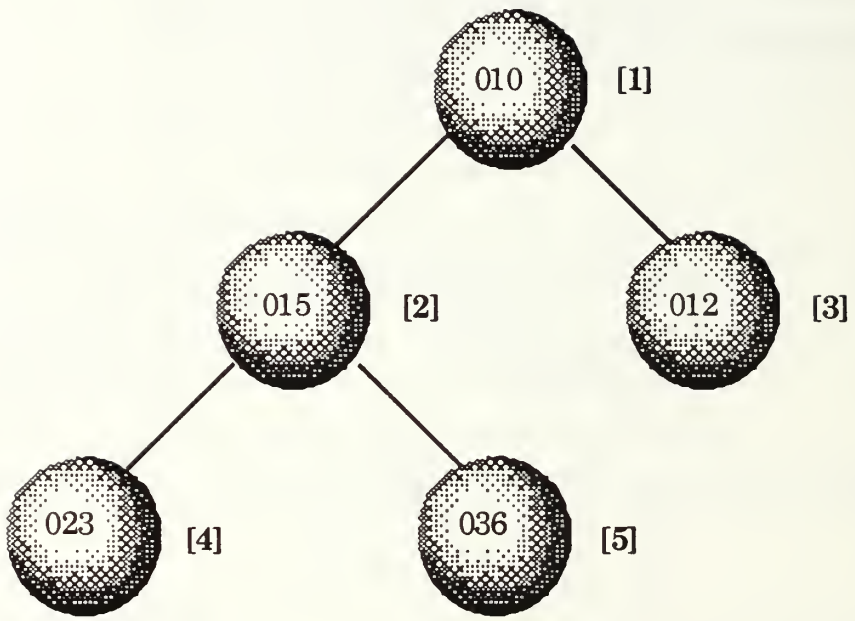


Figure 3.14 Heap After Element 004 is Dequeued

```
procedure siftdown(int : position)      /* move node down to satisfy        */
 begin                                  /* heap relative to its descendants */
 var
  int :        i,j;
  RQNode       save;
  boolean      finished;

   i = position;
   j = 2*position;
   save = RQNode[position];
   finished = FALSE;

   while(j<= RQSize) and not finished do      /* while there are children */
    begin                                     /* and position is not found */

         /* if there are two children - select the smaller */
    if(j < RQSize) and (RQNode[j].JobNumber > RQNode[j+1].JobNumber)
     then j = j + 1;

        /* if the position is found then finished  */
    if(save.JobNumber  <= RQNode[j].JobNumber)
      then finished = TRUE;
    else
     begin                      /* if not - move everything up and try again */
       RQNode[i] = RQNode[j];
       i = j;
       j = 2*i;
     end;
    end;
  RQNode[i] = save;
end;
```

Figure 3.15 Siftdown Routine Moves Elements Towards the Leaf Nodes [Ref. 5 : pp. 243-245]

than to move around an entire data structure almost twenty times the size of the pointer. However, using pointers without dynamic memory allocation meant having to keep track of the free space in the block of memory assigned to the nodes holding the data. Having to keep track of free space in the queue was additional overhead but nowhere near the overhead of moving entire nodes.

6. Purge Routine

Another requirement of the operating system is the need for a Purge routine. Originally, it was thought that a job could wait in the queue until it was processed and at the time of its processing it could be determined if the job was still valid or not. However, every twenty milliseconds new jobs will be added to the queue. Without a tremendous amount of memory, all the jobs enqueued cannot be allowed to stay in the queue indefinitely. The Purge routine purges or removes jobs that are no longer valid. Based on whether or not the valid bit is set, the job is technically "dequeued" from the middle of the queue. The job is dequeued because the Purge and Dequeue functions are so similar, but Dequeue is restricted to the top job in the queue. Figure 3.16 shows the flow diagram for the Purge module.

The Purge routine shown in Figure 3.17 is essentially the same as the Dequeue routine with the job being removed from the middle of the queue. Figure 3.18 shows the same integer heap as in Figure 3.14, but the elements 012 and 015 have been purged and the heap restructured. Since the Siftdown routine usually moves an element a lesser distance from its old location than from the root to the leaf node, the effort required to purge a single job from the queue is less than or equal to $0(\log_2 n)$.

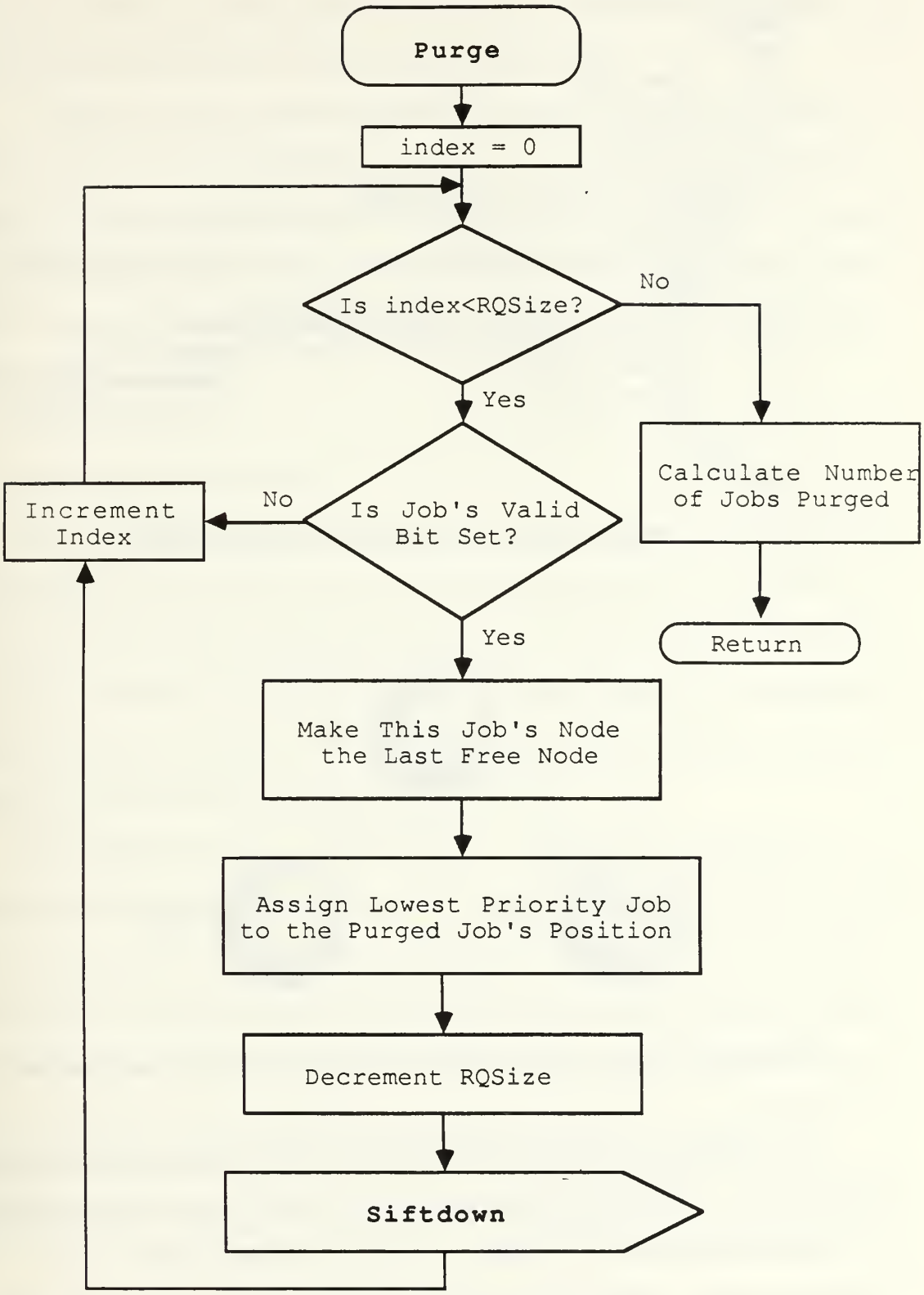


Figure 3.16 Purge Module

```
procedure Purge
  begin
  var
   int   index,temp;

   temp = RQSize;

     for index =1 to index <= temp do
       begin
        if( JobNumber.validbit)                    /* is Valid bit set? */
         begin
          RQNodePtr[index] = RQNodePtr[RQSize];
           /* put lowest job position in place of purged job */
          RQSize = RQSize - 1;                     /* decrease RQ Size   */
          siftdown(index);                         /* restructure heap   */
         end
       end;
  end;
```

Figure 3.17 Purge Routine to Remove Jobs.

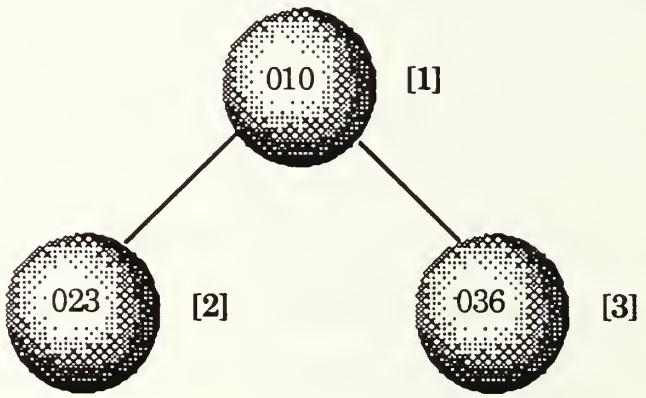


Figure 3.18 Heap after elements 012 and 015 have been Purged

The majority of the operating system's time is spent managing the Ready Queue. The beginning of every minor cycle is used to schedule tasks into the queue and to purge tasks no longer considered valid. The scheduling routine calls Enqueue for each job it adds to the queue. The

Central Task Processor's dispatch module calls the Dequeue function upon completion of a routine to remove it from the queue.

There are some smaller modules also associated with the Ready Queue whose code is contained in the appendix. The modules full and empty return a boolean based on the status of the queue. Create initializes the queue by setting the size of the queue equal to zero and setting up the next free node pointers. Create will be discussed in greater detail in the initialization section.

F. THE TASK SCHEDULER

The Task Scheduler is responsible for the scheduling of jobs at the beginning of each minor cycle. The Task Scheduler uses the Enqueue routine to place jobs in the Ready Queue and then increments the frame counter to indicate a new minor cycle. The Task Scheduler is called from and returns to the Cycle Interrupt Handler. Figure 3.19 shows the flow diagram for the Task Scheduler.

The Task Scheduler is responsible for determining which jobs are scheduled for a particular frame. Jobs are classified by how often they must be scheduled. The job numbers are located in arrays with an index, an offset and a constant for the array used in determining which jobs will be scheduled for a frame. There are four job arrays in the implementation of the operating system. The four arrays are lists of jobs to be scheduled every frame, every other frame, every fourth frame or every eighth frame. The frame counter is masked to determine an offset from the base pointer for

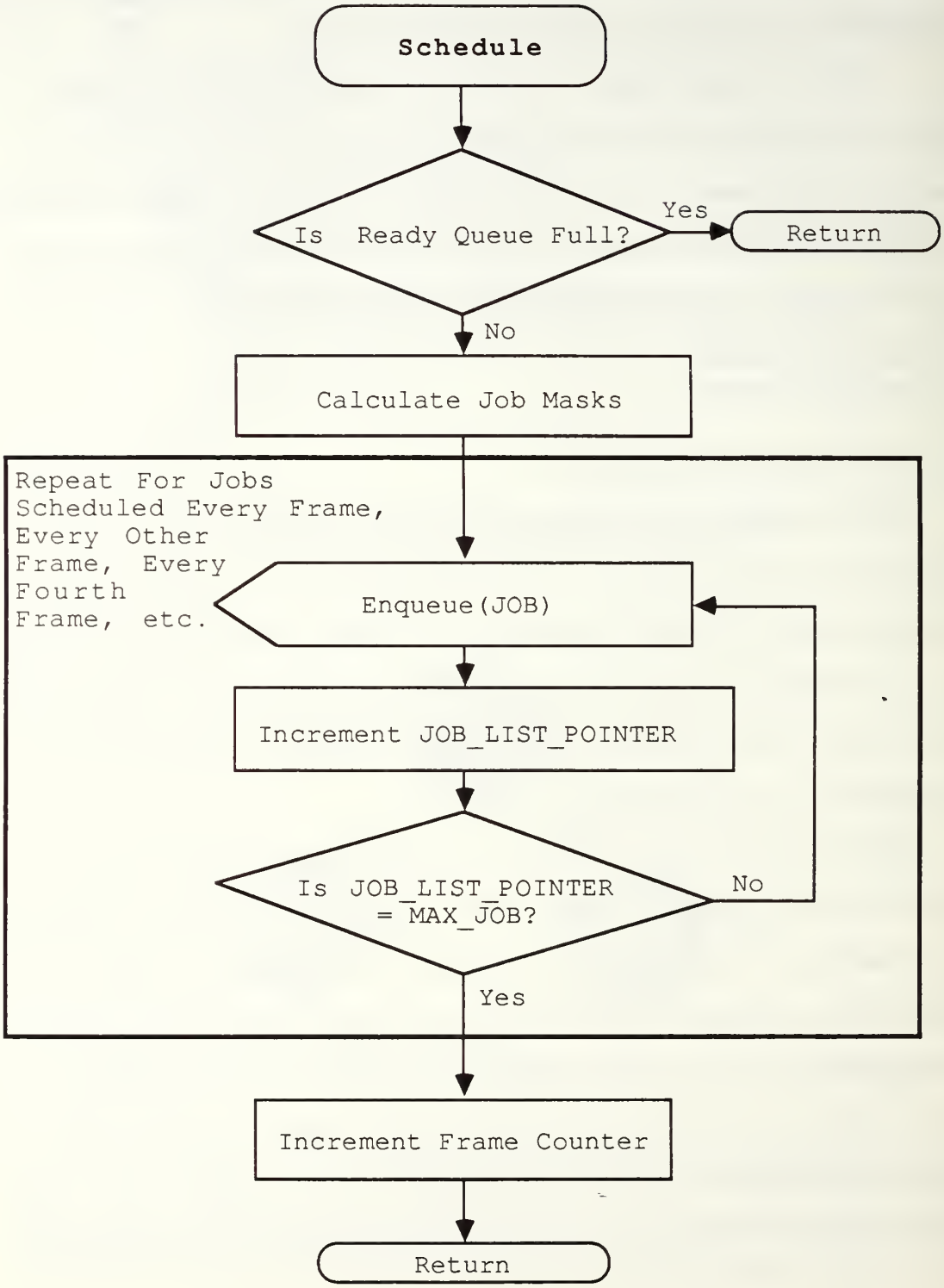
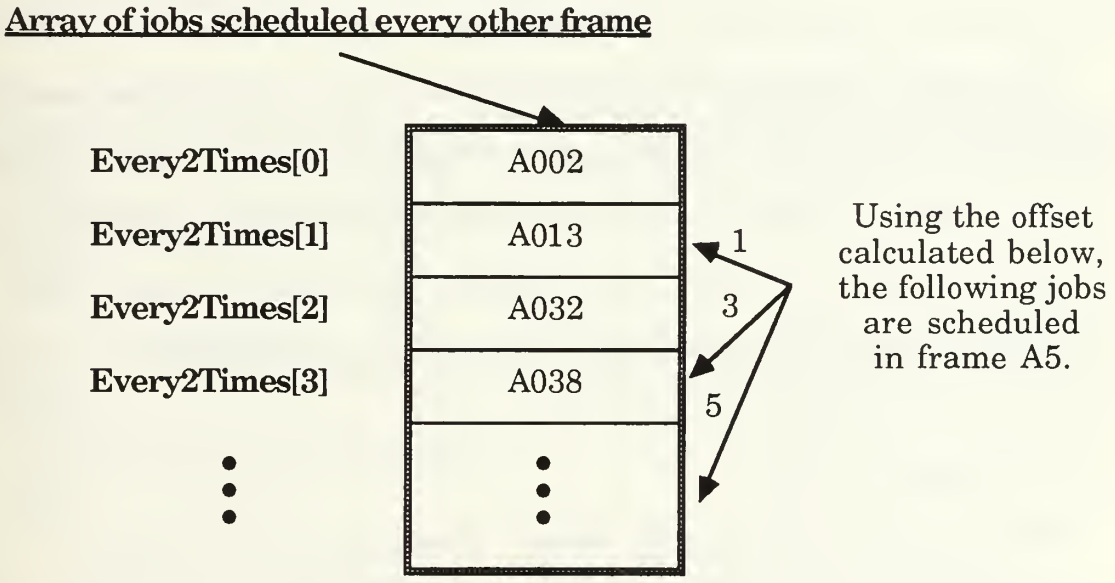


Figure 3.19 The Task Scheduler

each array. The masking of the frame counter is accomplished by performing a "bitwise and" operation with the frame counter and a constant for the array. The constant used to mask the frame counter is the frequency of how often the jobs are to be scheduled minus one. Along with the offset for each array is the constant stride factor that is the job scheduling rate. For example, jobs to be scheduled every fourth frame have a constant stride of four. The constant stride for an array is added to the offset computed from the Frame Counter and this sum is used as the index for the array of the first job to be enqueued. The constant stride is added to the index in a loop until the limits of the array or the maximum length of the list is reached and the next array is processed. This is illustrated in Figure 3.20.




```
Mask for Jobs Scheduled Every other Frame = 1
  Frame Counter = 00A5  Constant Stride = 2
Offset for this Frame is 0001 bitand 00A5 = 0001
```

Figure 3.20 Sample of the Job Mask

The "C" programming language is particularly useful for executing routines like the Task Scheduler. In Figure 3.21, a pseudocode routine for scheduling is shown. Comparing the routine in Figure 3.21 to the "C" implementation located in Appendix A shows how much more elegantly the scheduling routines can be coded in "C". Most languages do not have the "bitwise and" capability and many cannot easily loop through an array with a constant stride.

The scheduling of a set amount of tasks on a regular interval is essential to the design of the operating system presented in this thesis. The types of jobs and their execution length will determine which jobs and how many jobs can be scheduled into a single minor cycle. A desired result is an equal distribution of processing time over all cycles. It will be up to the programmer who codes the job to determine whether or not a job can be distributed over many minor cycles and thus form a major cycle for the job.

The length of the jobs to be scheduled and their execution frequency will also have a direct impact on the length of the minor cycle. The length was originally set at twenty miliseconds but this may be varied. There are other factors that may affect how long the minor cycle can be, however, these factors are mainly dependent on the application of the system.

G. INITIALIZATION

The kernel of the operating system is responsible for the system initialization on power up. The peripheral chips which support the microprocessor must all be initialized. There are several areas of memory, specific to the microprocessor, which must be set up prior to execution of

```
procedure Schedule()
 begin

JobMask2 =  FrameCounter bitand  1; /* Calculate job masks for    */
JobMask4 =  FrameCounter bitand  3; /* those tasks which are not */
JobMask8 =  FrameCounter bitand  7; /* scheduled every frame     */

index = 0;                          /* No offset required for these jobs    */
while index < MAXEVERY1 do          /* enqueue those that are scheduled */
 begin                              /* every frame (minor cycle).       */
    enqueue(Every1Time[index]);
    index = index +1;
 end;

index = JobMask2;                   /* start at offset for this frame and   */
while index < MAXEVERY2 do          /* enqueue those that are scheduled */
 begin                              /* every other frame (minor cycle).  */
    enqueue(Every2Time[index]);
   index = index +2;
 end;

index = JobMask4;                   /* start at offset for this frame and   */
while index < MAXEVERY4 do          /* enqueue those that are scheduled */
 begin                              /* every fourth frame (minor cycle). */
    enqueue(Every4Time[index]);
   index = index +4;
 end;

index = JobMask8;                   /* start at offset for this frame and   */
while index < MAXEVERY8 do          /* enqueue those that are scheduled */
 begin                              /* every eighth frame (minor cycle). */
    enqueue(Every8Time[index]);
   index = index +8;
 end;

FrameCounter = FrameCounter + 1; /* increment frame counter */
end;
```

Figure 3.21 Schedule Routine

the operating system's normal mode. The interrupt vectors and trap tables, common to most microprocessors, must be filled with the proper memory locations so that once the operating system has completed the set up routine, all interrupts may be serviced.

Figure 3.22 shows a limited overview of the major tasks which must be done upon start up. Most of these initialization routines are written in assembly language to facilitate the access to the hardware and to speed up the process. The operating system described in this thesis was developed on a microprocessor system that the had already completed these vector initializations using its own operating system. In order to test the operation of the operating system developed in this thesis, some of these vectors had to be replaced with vectors to the routines in the thesis operating system.

The Ready Queue must be set up during the initialization phase. Aside from allocating the space for the queue, the integer pointers in the nodes have to be initialized. The nodes are an array of data structures and the pointer to the next free node is the array subscript of the next free node. The next free node variable in each node data structure is initially filled with the next consecutive node. The free space is essentially a linked list with array subscripts used for pointers instead of addresses. The storage for the array of nodes and node pointers is allocated when the arrays are declared as global variables in the "C" header file. The header file, located in Appendix A, contains the global variable storage allocations for the entire operating system.

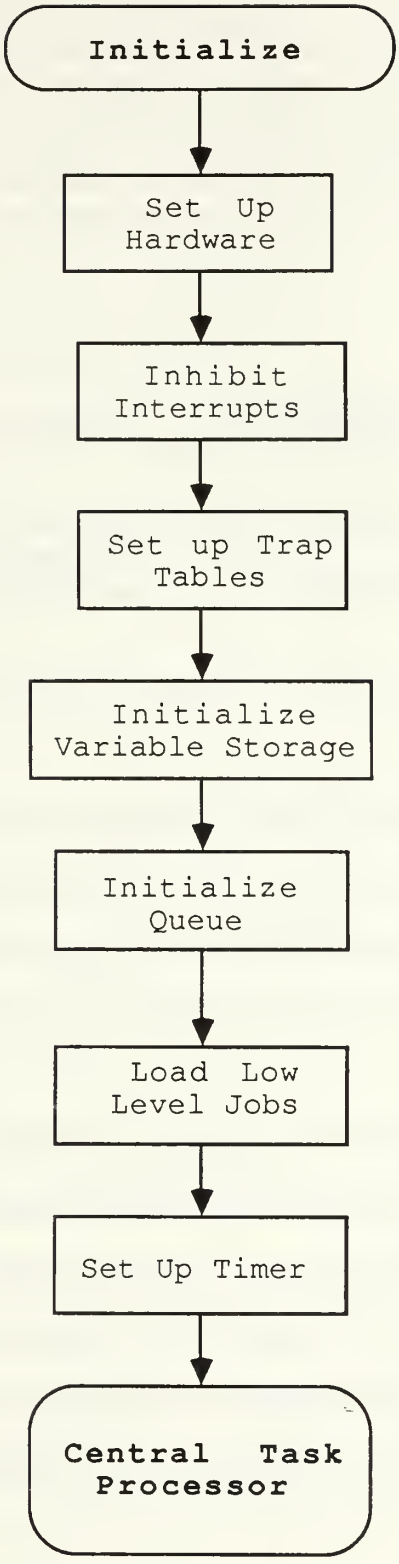


Figure 3.22 Initialization of the System

The array of pointers and the array of nodes must be tied together during initialization by setting the first pointer to be equal to the address of the first node. The process which accomplishes the initialization of the Ready Queue is the routine Create shown in Figure 3.23 below.

```
procedure create()
 begin
  RQSize = 0;                    /* initialize queue to empty state */

  for i= 0 to i < RQMAXSIZE  do     /* initialize free node pointers */
    RQNodes[i].NextFreeNode = i+1;    /* using array indices */

  RQNodePtr[0] = &RQNodes[0]; /* Point array of pointers to the */
  firstFree = 1;                      /* contiguous block of memory */
  lastFree = RQMAXSIZE - 1;       /* set globals for first & last free */
end;
```

Figure 3.23 Create the Ready Queue

The next phase of initialization is to load a low level job, a small monitor, into the queue. The monitor will never be removed from the queue since it never completes its execution and never returns back to a calling program. The monitor is assigned the lowest priority of any job that could be scheduled.

After loading the monitor, the timer is initialized and started and the control of the program is given to the Central Task Processor for execution of the first job in the Ready Queue. The timer used for this thesis was a SY6522 Versatile Interface Adapter. The initialization consisted of programming one of the two sixteen bit interval timers located on the SY6522 to interrupt the microprocessor at the beginning of every minor cycle.

In the development of a single microprocessor system, there are several initialization steps which can be overlooked. The most significant initializations to be done in a multi-processor system is the establishment of inter-processor communications and synchronization of the processors. These two vital tasks, communications and synchronization, are the responsibility of the operating system kernel.

IV. PERFORMANCE MEASUREMENTS

A. BACKGROUND

The operating system was implemented on a MC68000 based personal microcomputer to allow convenient testing of the modules. The Ready Queue was written and tested first because it was believed that manipulating and maintaining the Ready Queue would be the most time critical area for the operating system. This belief was based on the fact that at the beginning of every minor cycle the scheduler would have to enqueue a preset number of jobs and at the end of the minor cycle some jobs might have to be purged.

The development hardware used to do the implementation and testing was not optimal. Not only is a personal microcomputer not designed for system development (lack of a dual bus system, separate exception tables, hardware breakpoints, trace capture, etc.) but a personal microcomputer usually has several features which slow the speed of the overall system. A personal microcomputer also uses shortcuts in the hardware design to save on the final cost of the product. There were several system tasks important to the host microcomputer being carried out at the same time as the testing of the operating system routines. The details of the hardware layout are in Appendix C.

As an example to illustrate some of the hindrances in the execution of the test routines, the microcomputer required a screen refresh every 16.67 milliseconds. A screen refresh is a level one interrupt and it was not

feasible to replace all level one interrupt routines with the routine for the timer interrupt which shared a level one status. The computer used a secondary vector table to route the level one interrupt routines to their proper handlers. The Cycle Interrupt routine was vectored from the secondary vector table.

Even though the test environment was not optimal, measurements were taken for execution times of the different ready queue routines for comparative purposes. Measurements were not taken of the routines written in assembly language. The assembly language routines were so short that a more accurate indication of the assembly language routine's execution times could be calculated adding up the individual instruction times. The individual instructions execution times are available in the Appendix of Reference [7]. The minor cycle used to test the operating system was sixty milliseconds. The extra time was required due to the operating environment.

The efficiency of the compiler is another factor to consider when evaluating the execution times of various routines. The time for execution of a routine may vary for different compilers. A routine may execute at a slower speed if the compiler does not optimize the final object code by making several compilation passes and stripping dead code. The compiler used in the generation of the operating system in this thesis was the Mac C™ compiler by Consulair Corporation. This compiler should not be considered optimal. It makes only one pass of the "C" code for compilation, one pass for preprocessor commands and one pass for assembling in-line assembly language code.

There were benefits in developing the operating system on a microcomputer. One benefit is that the software could be tested on a proven hardware system eliminating the conflicts of bugs from both hardware and software. Another benefit is that a run time library need not be built in order to use the higher level routines in "C", such as input from the keyboard and output to the screen. The operating system was developed to be used on another MC68000 system but an emphasis was placed on portability .

Many of the routine tasks that would normally have to be coded, as in the initialization routines, had already been accomplished by the personal microcomputer's own operating system. The microcomputer environment, while not optimal for evaluation of the final product, can be used in the development of systems software.

B. COMPARISONS

1. Scheduling

The Scheduler is called at the beginning of every minor cycle. The Scheduler then calls the routine Enqueue several times depending on how many jobs are being scheduled in the frame. The execution time of the scheduler is directly related to the efficiency of the Enqueue routine. Other than enqueueing jobs, the Scheduler's only other tasks are to calculate the job offsets for each job list and increment the Frame Counter.

Since the order of magnitude for a queue insertion is $O(\log_2 n)$, the order of magnitude for inserting n items in the queue should be $O(n \log_2 n)$.

The graph in Figure 4.1 shows the execution times for loading the ready queue using the Schedule routine three different times. The schedule routine inserted eight jobs per minor cycle into the ready queue. The queue was essentially full after six minor cycles since no execution of tasks or purges were taking place. The time to schedule eight jobs does not appear to increase substantially as the queue gets full, but it does increase.

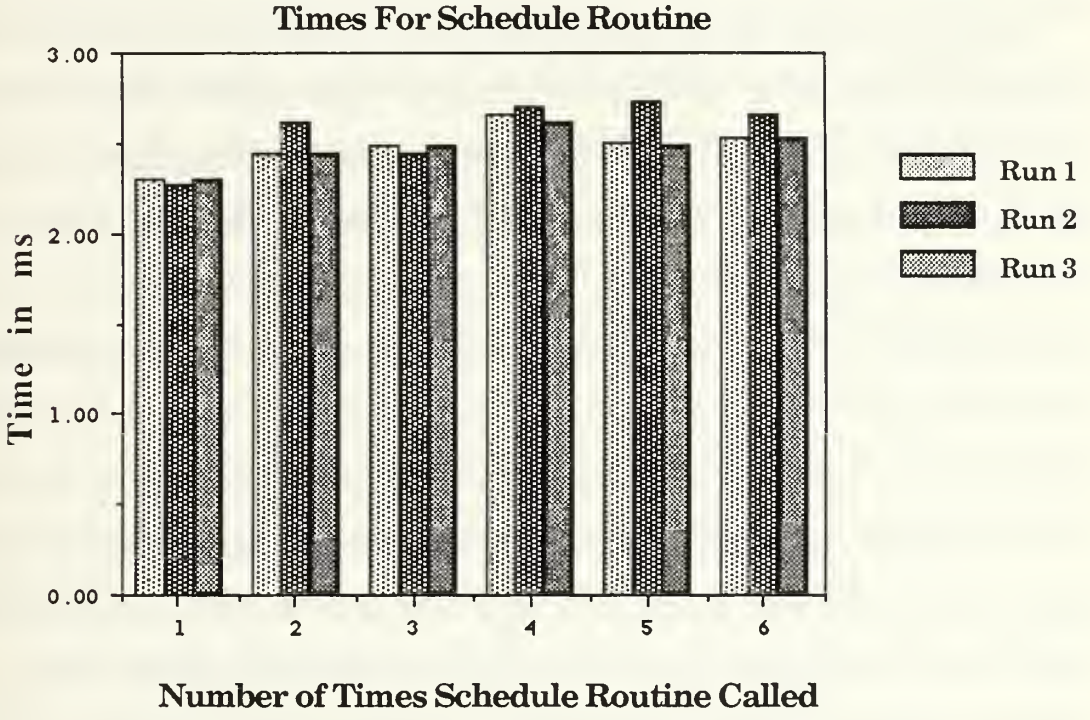


Figure 4.1 Schedule Routine Performance

The numbers used to generate Figure 4.1 are found in Appendix D. It would appear that it takes on the average approximately 2.5 milliseconds to schedule eight jobs into the ready queue. Not allowing for the mask calculations or incrementing the frame counter, it takes about 0.3 milliseconds to schedule one job into the ready queue. This is not an

acceptable figure for a twenty millisecond minor cycle. However, the measurements taken should be considered relative for comparison to other measurements taken on the same system, due to the environment, and do not reflect the true performance of the operating system in a dedicated system.

2. Purging

Purging of the Ready Queue must take place prior to the scheduling of new tasks in a new minor cycle to make room for the new jobs. The purge operation will usually not remove the same number of jobs each minor cycle since not all jobs in the Ready Queue are purgeable. There may be a minor cycle in which two or more items are purged. The most likely occurrence is the case of one or no jobs purged per minor cycle.

A major factor affecting the execution speed of the purge routine is the location of the job in the heap which is the Ready Queue. A job which is higher in the Queue with a higher priority and is closer to the root will take longer to purge than a job located in a leaf node. The location dependence of purge time differences is due to the siftdown routine called to restructure the heap. The job from the lowest location in the heap is placed in the location of the old purged job, therefore, the closer the purged job to the leaf node, the less "sifting down" that has to be done to put the heap back in order.

Since there are so many factors affecting the execution time of the purge routine, an in-depth analysis was not done. It was believed that due to the nature of the environment, the measurements would not be accurate enough to be of any significance. The factors involved in a call to the Purge

routine include the size of the Ready Queue when Purge is called and the number of purgeable jobs in the queue and their locations. It should be noted that the bound on the performance of a single Purge is of O(log2 n), the same as a dequeue or a purge of the top element in the queue.

3. Enqueueing

The performance of a single insertion into the queue is of O(log2 n). The main factors affecting the execution time of the enqueue routine are the size of the queue and the priority of the job being inserted. Only a single entry at a time is made into the queue. The Enqueue routine calls the Siftup routine to restructure the queue after a new element has been added. It should be noted that the order of a single insertion is an upper bound to the execution time of a single insertion. If the new element happens to be the lowest priority job in the queue then no restructuring will be necessary and the time to execute the Enqueue routine will be less than O(log2 n).

Since the execution time of a single queue insertion is dependent on the size of the queue as well as the priority of the job being inserted, consideration of the job's priority must be taken into account when deciding which jobs will be scheduled into the different minor cycles. If many high priority jobs are enqueued at one time, it could conceivably slow the Scheduler down so that there would be as much time left in the minor cycle for job execution.

The performance of the enqueue routine is encompassed by the performance measurements of the the scheduler. Measurements taken of individual enqueueing operations would be meaningless unless the heap was shown before and after each insertion and the job placement in the

queue was monitored carefully. Even keeping track of the queue structure, there are the other variables of the queue size and the priority of the job to be inserted.

4. Dequeueing

The Dequeue routine is not called until the job that is being removed from the queue has completed execution. Dequeue calls the Siftdown routine to restructure the heap and since the lowest priority job is placed in the highest priority position, the time to perform a removal from the queue is always O($\log_2$ n). Therefore, the main factor in affecting the performance of the Dequeue routine is the size of the queue. The illustration of the Dequeue routine's dependency on the size of the queue is illustrated in Figure 4.2. A logarithmic interpolation curve is used in Figure 4.2 to emphasize the log relationship between the performance of the Dequeue routine and the size of the queue.

C. CONCLUSIONS

The operating system was not tested in an environment that allowed for an accurate performance evaluation. The routines that were tested appeared to do relatively well. The times are inflated due to the conditions, however, the times appear to reflect the proper operation of the queue management routines.

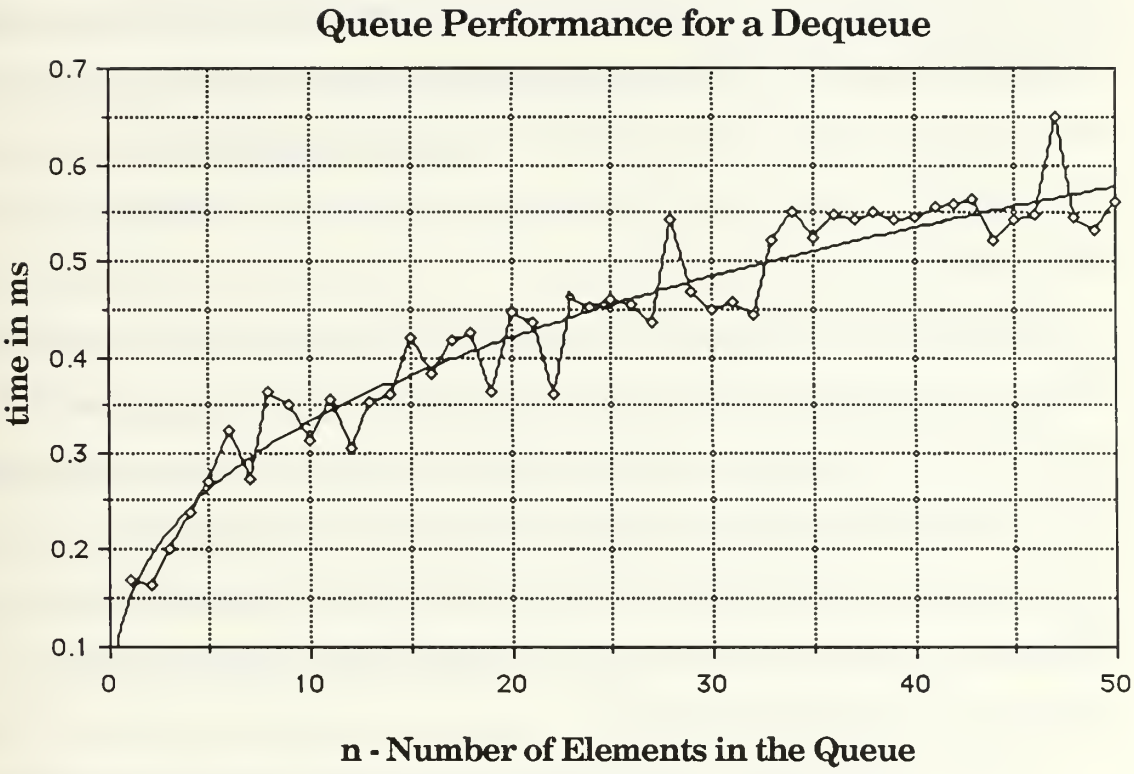


Figure 4.2 Performance of the Dequeue Routine

V. CONCLUSIONS

A. SUMMARY OF RESULTS

The principal goal of this thesis was the development of a kernel for a real time fault tolerant operating system. The modules which make up the operating system kernel were designed and tested individually before being combined to form the bulk of the operating system kernel. The objectives of the thesis were met to the extent that an operating system kernel was designed for a real time fault tolerant environment and implemented on a MC68000 based microcomputer.

The real time requirement was described by the ability of the operating system to respond to events as they occurred. The interrupt handler of the operating system meets this requirement. If a real time request interrupts the processor with a high enough priority to be placed on the top of the Ready Queue, it is serviced immediately. There are other external influences which will need to be serviced in real time but that are not handled by the interrupt handler.

A major requirement which must be addressed before this operating system can be implemented in a multiprocessor environment is the ability to communicate with the other processors. This will be an operating system kernel function with real time requirements.

The use of the A-Line trap to monitor the execution of all jobs was originally intended to enhance the fault tolerance of the operating system without impeding operating system efficiency. The assignment of an

A-Line and priority to each routine ensures that the operating system is aware of the status of all active jobs. However, the combination of the Central Task Processor and the A-Line trap results in redundant management. The A-Line trap actually slowed the performance of the operating system by adding extra instructions to handle the A-Line exception in order to execute a job. A table of contents could be used with the same effect as the A-Line trap and with better efficiency. The jobs' starting addresses could be located in a table of contents and still be prioritized by job number in the table. Using a table of contents eliminates the need to go through the A-Line trap handler.

The use of "C" is required to maintain portability between systems but the object code produced by the "C" compiler may not be optimal. Most "C" compilers contain a feature that allows the generation of an intermediate assembly language file during compilation. The intermediate assembly language code could be optimized for time consuming operations performed by the operating system. The reason for optimization is that many compilers inherently include dead code and duplicate instructions in the object code created as a result of the compilation. The removal of the dead code from the intermediate assembly language file should help the operating system achieve better performance.

A hardware system for the design and implementation of the operating system is required. Without the intended microcomputer system available for the operating system, many routines dealing with hardware specific functions of the operating system could not be written and tested.

B. RECOMMENDATIONS FOR FURTHER RESEARCH

The operating system kernel developed in this thesis will need to be expanded upon as more sophisticated functions are required. The ability to send and receive messages between processors in a multiprocessor system will require the support of input/output functions. The addition of a full function input/output capability and additional processors in the system will produce a requirement for semaphores to monitor the use and availability of the input/output resources. The addition of semaphores means the requirement for an additional queue for jobs which are waiting for blocked resources. There will then be a requirement for a Ready Queue and a Waiting Queue.

Further research into the operating system will require a proper development station. The personal computer used to develop the operating system kernel was not a true development environment. A hardware design that is intended for a real time fault tolerant environment should be used to test future versions of the operating system. The use of the proper hardware will help to increase the accuracy of the results for the test and evaluation of the operating system routines.

APPENDIX A

C AND ASSEMBLY LANGUAGE SOURCE CODE LISTINGS

This Appendix contains the source code modules required for the operating system. The listings start with the header file which was included in each source code module. The "Main" function, required in a C language program, is in the file OSTest.c and has been made as generic as possible considering the development environment. All the other listings follow. A block diagram illustrating the exection flow between modules is loacted in Figure A.1

The C programming language was commented as much as possible to enhance readability. The standard C as professed by Kernighan and Ritchie in Reference [6] was used to insure portability. The modules are presented in the approximate order in which they are used. Each section denoted by a heading of a module name is a different file, compiled separately and later linked together.

The file OSTest.c was modified to test each module as it was written. Also some files were modified to use the SY6522 timer to time the performance of the routine and gather the data in Appendix D.

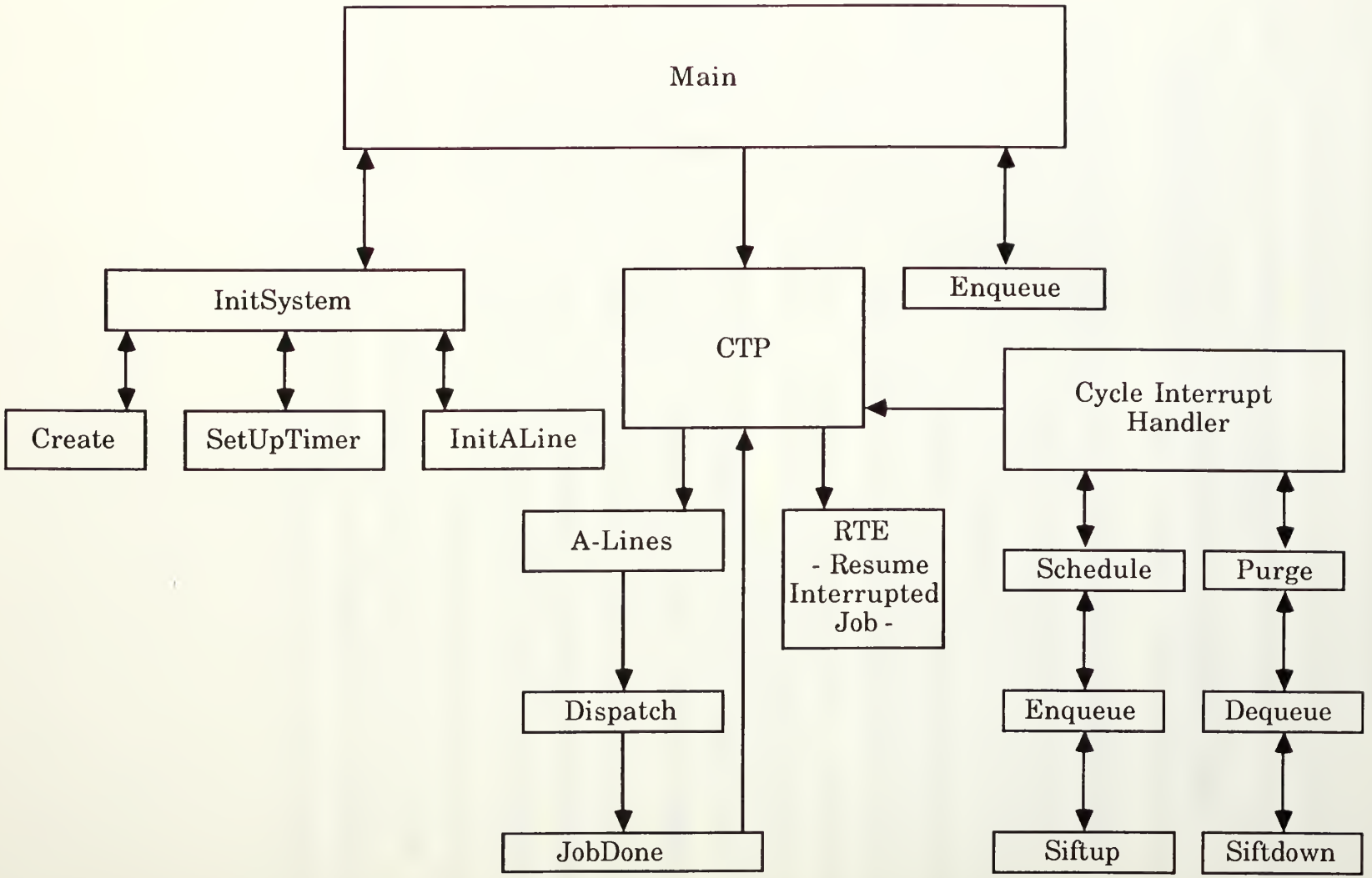


Figure A.1 Operating System Module Flow Diagram

```
****************** A_LINETEST.H ******************

/*  A-linetest.h,  header file for Operating System Global Variables  */

#include  "C_Heads.h"   /* include Macintosh stdio.h pre-compiled  */

#define       TRUE        1
#define       FALSE       0

#define RQMAXSIZE     50      /* max size of ready queue */

/* These constants are the max number of jobs that can be scheduled - */
#define  MAXEVERY1    5       /*  - every frame */
#define  MAXEVERY2    4       /*  - every other frame */
#define  MAXEVERY4    8       /*  - every fourth frame */
#define  MAXEVERY8    16      /*  - every eighth frame */

#define MAXJOBNUMBER  1023   /*  max total number of jobs = 03FF */

struct _RQInfo                      /* the Ready Queue Node */
   {
   short              JobNum;          /* A-line trap number    */
   long               FrameStart;      /* Frame Scheduled in    */
   short              SerialNum;       /* Index for TOC and Q   */
   int                NextFreeNode;    /* index number of next  */
   long               Data0;           /* all Data and Address  */
   long               Data1;           /* Registers stored here */
   long               Data2;
   long               Data3;
   long               Data4;
   long               Data5;
   long               Data6;
   long               Data7;
   long               Addr0;
   long               Addr1;
   long               Addr2;
   long               Addr3;
   long               Addr4;
   long               Addr5;
   long               Addr6;
   long               Addr7;       /* stack points at SR and PC */
   };

#define RQNode    struct _RQInfo     /* define Node */
typedef RQNode    *RQPtr;            /* define pointer to a node */
```

```
#ifdef MAIN                    /* if compiling the main() program */

int             i,j,k,l,m;                      /*  general indices list */
RQNode          RQNodes[RQMAXSIZE];     /* make space for Nodes    */
RQPtr           RQNodePtr[RQMAXSIZE];   /* make space for pointers */
unsigned char   SerialList[MAXJOBNUMBER]; /* serial number array */
int             RQSize                  /* The size of the ready queue */
 int            firstFree,lastFree;     /* for keeping track of free space */
unsigned long   JobMask2,JobMask4,JobMask8; /* masks for scheduler */
short           Every1Time[MAXEVERY1];  /* These are the arrays of  */
short           Every2Time[MAXEVERY2];  /* job numbers, classified */
short           Every4Time[MAXEVERY4];  /* into lists of how often  */
short           Every8Time[MAXEVERY8];  /* they are scheduled.      */
unsigned long   FrameCounter;           /* global 32 bit value */
extern          RQPtr  serve();         /* if serve returns a value */
long            GlobalRegs[16],MacTraps;    /* These variables were  */
short           TopJob,PresentJob;          /* for use in ASM code  */
long            Regs,OldJob;

#else     /* all global variable storage space is defined in the main file */

extern          int             i,j,k,l,m;

extern      RQNode          RQNodes[];
extern      RQPtr           RQNodePtr[];
extern      unsigned char   SerialList[];
extern      int             RQSize,
extern      int             firstFree,lastFree;
extern      unsigned long   JobMask2,JobMask4,JobMask8;
extern      short           Every1Time[],Every2Time[];
extern      short           Every4Time[],Every8Time[];
extern      unsigned long   FrameCounter;
extern      long            GlobalRegs[],MacTraps;
extern      short           TopJob,PresentJob;
extern      long            Regs,OldJob;

#endif
```

********** OSTEST.C **********

```
/* This  program contains the main() routine which is responsible for the */
/* initialization of the queue and all other global variables. It is      */
/* responsible for the set-up of the operating system and the loading of */
/* the monitor prior to the calling of the Central Task Processor. While */
/* it contains a never ending loop, this loop will never be executed   */
/* more than once.  Once control is passed to the Central Task Processor, */
/*  it never returns here. */
```

```
#define MAIN
#include "A-linetest.h"    /* include header file  */

InitSystem()
{
FlushEvents(everyEvent);  /* cleans out Event Queue on Macintosh  */

create();                 /* Creates the ready queue  */
FrameCounter = 0;         /* Insure frame counter starts at zero */
SetUpTimer();             /* Set up SY6522 VIA for cycle interrupts*/
InitALine();              /* Load our trp handler in low memory */
}


main()
{
InitSystem();             /* do initialization  */

enqueue(0xF003);          /* enqueue the monitor */

#asm                      /* in-line assembly allowed */
XREF  doSchedule

LEA   doSchedule,A0       ; Loads our routine into dispatch table
MOVE.L     A0,$192        ; Level 1 Interrupt - secondary table
#endasm

while (TRUE) {      /* endless loop */
  CTP();            /* Let Central Task Processor take over */
  }
}                   /*end of main() */

SetUpTimer()
{
#asm
VIA          EQU  $1D4    ; VIA base address [pointer]
vIER         EQU  $1C00   ; INT. ENABLE REG.

MOVEA.L   VIA,A0
BCLR      #6,vIER(A0)          ; disable timer1 interrupt until ready

#endasm
}
```

********** CTP.ASM **********

```
/*  The Central Task Processor was written entirely in assembly       */
```

```
/*  for ease of accessing the A-Line trap and for speed.                */

INCLUDE M68KLIB.D                    ;   required assembly library

 XREF RQNodePtr                      ;   only external variable

 CTP:
 MOVE.L    RQNodePtr+4(A5),A0        ; retrieve node pointer
 MOVE.W    (A0),D0                   ; retrieve job number
 AND       #$800,D0                  ; test to see if new job or not
 BEQ       newjob                    ; if new use A-line

   MOVEM.L   12(A0),D0-D7/A0-A7      ; restore regs
   RTE                               ; continue interrupted job

newjob:
   LEA       Atrap,A1                ; Load address space
   MOVE.W    (A0),(A1)               ; move A-Line there
Atrap:
   DC.W      $4E71                   ; Reserved word for A-Line
END
```

********** READYQ.C **********

```
/*  ReadyQ.c  -  This module includes all the functions required */
/*  for set up and maintenance of the ready queue.          */

#include  "A-linetest.h"          /* includes all global variables */

/* --------------------- Enqueue --------------------------*/

enqueue(jobNum)  /* add a job to the queue   */
short   jobNum;
{
unsigned short     index, shifted;

 if(RQSize == RQMAXSIZE)                    /* normally would want to */
    return;                                 /* set an alarm here */

   RQSize += 1;                                  /* increase RQ Size */
   RQNodePtr[RQSize] = &RQNodes[firstFree];      /* get free space for job */
   firstFree = RQNodes[firstFree].NextFreeNode; /* change first free space */
   RQNodePtr[RQSize]->JobNum = jobNum;       /*  Load Job Number and */
   RQNodePtr[RQSize]->FrameStart = FrameCounter;   /* Frame Number  */

   index = (jobNum & 0x03FF);            /* Index for serial number list  */
```

```
  shifted = index << 4;                    /* put jobnum in High  byte 1/2  */
                /* Serial number = | Jobnum | serial num|   */
   RQNodePtr[RQSize]->SerialNum = SerialList[index] + shifted;
   SerialList[index] +=1;                  /* inc serial num for that job*/
   siftup(RQSize);                         /* Move it towards Root */
}

/* ---------------------- Serve --------------------------*/

RQPtr  serve()       /* Remove the top job from the queue. */
{
RQPtr  JobPtr;
short index, shifted;
int    next;

   JobPtr = RQNodePtr[1];                  /* Retrieve the highest Priority    */
   next = JobPtr - &RQNodes[0];            /* make the now vacant space...  */
   RQNodes[lastFree].NextFreeNode = next;  /* the last free space */
   lastFree = next;
   RQNodePtr[1] = RQNodePtr[RQSize];       /* Put low job at the top */
   RQSize -= 1;                            /* decrease RQ Size  */
   siftdown(1);                            /* restructure heap  */
   return(JobPtr);
}

/* ---------------------- Create --------------------------*/

create()       /*    Sets up the global variables for the ready queue.       */
{
  RQSize = 0;                        /*    Initialize queue to empty state      */

  for(i= 0; i < RQMAXSIZE -1; i++)      /* Initialize free node pointers  */
    RQNodes[i].NextFreeNode = i+1;

 RQNodePtr[0] = &RQNodes[0];          /* Point array of pointers to the    */
 firstFree = 1;                       /* contigous block of memory        */
 lastFree = RQMAXSIZE - 1;            /* set globals for first & last free   */
}

/* ---------------------- Siftup --------------------------*/

siftup(pos)                           /* Move towards Root */
int    pos;
{
int    j,k;
short serialnum;
```

```
   serialnum = RQNodePtr[pos]->SerialNum; /* retrieve the serial number*/
   RQNodePtr[0] = RQNodePtr[pos];          /* save the present node pointer */
   k = pos;                                /* save the present position      */
   j = (int)(pos/2);                       /* get the first parent's postion  */

    while(RQNodePtr[j]->SerialNum > serialnum)
     {
      RQNodePtr[k] = RQNodePtr[j];          /* move up one place       */
      k = j;                                /* and compare the child */
      j = (int)(j/2);                       /* with its' parent        */
      }

    RQNodePtr[k] = RQNodePtr[0];     /* location found move element there */
}

/* ---------------------- Siftdown ---------------------------*/

siftdown(pos)                        /* move node down to satisfy the    */
int     pos;                  /* heap relative to its descendants */
{
int     i,j;
RQPtr         save;
char   finished;

    i = pos;                         /* will want to start with parent       */
    j = 2*pos;                       /* and start with one of its children   */
    save = RQNodePtr[pos];           /* save new node until finished         */
    finished = FALSE;                /*  boolean to know when we're done */

    while((j<= RQSize) && (!finished))    /* while there are children */
     {

      /* if there are two children - select the smaller */
if((j<RQSize) &&(RQNodePtr[j]->SerialNum > RQNodePtr[j+1]>SerialNum))
       j += 1;

       /* if the position is found then it's finished */
       if(save->SerialNum  <= RQNodePtr[j]->SerialNum )
        finished = TRUE;
      else
       /* if not - move next node up and try again */
       {
          RQNodePtr[i] = RQNodePtr[j];  /* switch places */
          i = j;                        /* look at next parent */
          j = 2*i;                      /* and its' children     */
          }
     }
```

```
   RQNodePtr[i] = save;            /* Place new node in its proper place  */
 }

/* ---------------------- Empty --------------------------*/

empty()                 /* test for an empty queue        */
{
 int    RQempty;

 if(RQSize == 0)
   RQempty = TRUE;       /* either its empty  */
 else
    RQempty = FALSE;     /* or it isn't  */

 return(RQempty);        /* return boolean type answer  */
 }

/* ---------------------- Full --------------------------*/

full()                  /* test for a full queue  */
{
 int    RQfull;

 if(RQSize == RQMAXSIZE)
   RQfull = TRUE;            /* either its full  */
 else
    RQfull = FALSE;          /* or it isn't */

 return(RQfull);             /* return boolean type answer  */
 }

/* ---------------------- Purge --------------------------*/

Purge()
{
int     next,temp;

purgeNum = 0;               /* initialize number of jobs purged this time */
temp = RQSize;              /* needed for index  */

  for(i=1;i<= temp;i++)     /* Search only through actual size of queue  */
    {
     if(RQNodePtr[i]->JobNum & 0x0400)       /* if job is purgeable */
       {
          next = RQNodePtr[i] - &RQNodes[0]; /* making  new free space */
          RQNodes[lastFree].NextFreeNode = next;
          lastFree = next;                          /* make it the last free */
```

```
            RQNodePtr[i] = RQNodePtr[RQSize];  /* Put lowest job in its place */
            RQSize -= 1;                       /* decrease RQ Size   */
            siftdown(i);                       /* restructure heap   */
           }
      }
   purgeNum = temp - RQSize;          /* how many jobs were purged? */
}
```

********** SCHEDULE.C **********

```
/* Schedule.c contains the routines for the Cycle Interrupt Handler    */
/*  and the scheduler. */

#include "A-linetest.h"  /* include global variables */

/*  doSchedule is also known as the Cycle Interrupt Handler.     */
/*  This piece of code is the beginning of every new frame.      */
/*  On the Macintosh, this routine is in the level 0 interrupt   */
/*  table which is shared by several other devices.              */

doSchedule()
{
#asm
XREF Schedule,CTP

    MOVE        #$2700,SR               ; inhibit interrupts
    MOVE.L      RQNodePtr+4(A5),A0      ; get current job info
    MOVEM.L     D0-D7/A0-A7,12(A0)      ; store current registers
    MOVE.L      OldJob(A5),A1
    BTST        #6,$EFFBFE              ; is it timer 1?
    BNE         timer1                  ; yes do it
    JMP         (A1)                    ; OldJob is an Apple routine

timer1:
    MOVE.W      (A0),D0                 ; get top job number
    OR          #$0800,D0               ; Set rupt bit
    MOVE.W      D0,(A0)                 ; put new job num back
    JSR         Purge                   ; clean out Queue
    JSR         Schedule                ; schedule new frame tasks
    JMP         CTP
    RTS
#endasm

}
Schedule()                      /* routine to schedule jobs */
{
```

```
if(RQSize == RQMAXSIZE)
  return;                          /* Would normally trip an alarm */

JobMask2 = (FrameCounter & 1);     /* calculate job masks for those tasks  */
JobMask4 = (FrameCounter & 3);  /* which are not scheduled every frame */
JobMask8 = (FrameCounter & 7);

for(i=0; i<MAXEVERY1; i++)   /* enqueue those that are scheduled  */
  enqueue(Every1Time[i]);        /* every frame   */

for(i=JobMask2; i<MAXEVERY2; i+=2)   /* enqueue those that are      */
  enqueue(Every2Time[i]);                 /* scheduled every  other frame */

for(i=JobMask4; i<MAXEVERY4; i+=4)   /* enqueue those that are      */
  enqueue(Every4Time[i]);                 /* scheduled every fourth frame */

for(i=JobMask8; i<MAXEVERY8; i+=8)   /* enqueue those that are     */
  enqueue(Every8Time[i]);                 /* scheduled every eighth frame*/

FrameCounter += 1;                        /* increment frame counter */
}
```

**********A_LINES.C **********

```
/* A-Line Trap Handler */

#include "A-linetest.h"     /* Include file for global variables and defs */


/* InitALine  actually replaces the F-Line Trap Handler on the Macintosh*/
/* The A-Line Trap is used on the Macintosh, however the end result is */
/*  the same as they are both unimplemented instructions. */

InitALine()
{

#asm
    MOVE.L      $2C,MacTraps(A5)        ; Save old A-Line Handler
    LEA         ALines,A0               ; Load Trap Handling Routine
    MOVE.L      A0,$2C
#endasm

}


/* This is the actual routine vectored to ater the execution of a trap     */
/*  Since there will never be an RTE from this routine the stack must  */
/*  be adjusted prior to leaving the routine.                          */
```

```
ALines()            /* A-Line Trap Handling Routine  */
{

#asm
   XREF    Dispatch

   MOVE          #$2700,SR          ; insure no interruptsrupts
   MOVEA.L       $02(A7),A2         ; fetch PC and its contents
   MOVE.W        (A2),D2            ; Store trap in D2
   ADDQ.L        #$6,A7             ; adjust stack
   ANDI.L        #$03FF,D2          ; get last 10 bits of word-job number
   MOVE.W        D2,PresentJob(A5); store job number for dispatch table
   JMP           Dispatch           ; go to dispatch table
#endasm

}

/* The Dispatch Table is where the actual Job is vectored to.      */
/*  The job also will return here upon completion.          */

Dispatch()                  /* job dispatch table  */
{                           /* limited number of jobs version */

switch(PresentJob)          /* based on job from A-Line Trap Handler  */
       {

         case 1:
          Job1();           / * handle Job 1 */
          JobDone();        /* After Job is completed  */
         break;

          case 2:
          Job2();           / * handle Job 2 */
          JobDone();        /* After Job is completed  */
         break;

         case 3:
           monitor();       / * handle Job 1 */
         break;             /* Monitor is never completed  */

         default:
           JobD();          / * handle all the other jobs */
           JobDone();       /* After Job is completed  */
         break;
       }
 }
```

```
/* The JobDone routine is responsible for removing the job from the    */
/*  ready  queue by calling the Serve function. This routine should not */
/*  be interrupted. Control is passed on to the CTP after completion.    */

JobDone()
{

#asm
XREF CTP,Serve

MOVE        #$2700,SR          ; inhibit interrupts until done
JSR         Serve              ; Remove top job from ready queue
ADDQ.L      #$4,A7             ; adjust stack since there is no RTS
JMP         CTP                ; Go back to CTP for next job
#endasm

}


/*  The following are small sample jobs used to test the OS  */
Job1()
{
#asm
MOVE        #$2000,SR          ; required of all jobs written
MOVE.B      $EFE9FE,$FFCF7  ; read 6522 timer count lower byte
MOVE.B      $EFEBFE,$FFCF6 ; read 6522 timer count HIGHER byte
#endasm
}

Job2()
{
#asm
MOVE        #$2000,SR          ; required of all jobs written
MOVE.B      $EFE9FE,$FFCF9  ; read 6522 timer count lower byte
MOVE.B      $EFEBFE,$FFCF8 ; read 6522 timer count HIGHER byte
#endasm
}

JobD()
{
/* NULL Job  */
}
```

APPENDIX B

SOURCE CODE LISTSINGS FOR A PARTIAL MONITOR

This Appendix contains the source code modules for a partial monitor. The monitor was used as a low level job to test the operating system. It is a partial monitor in that it can only do memory display and memory modify. A full function monitor should aslo be able to display and modify registers, set breakpoints and allow the running of another program using a "Go" routine. All the functions which were not implemented are located in a stub module to complete the linking process. This should make the further refinement of the program easier as well.

The listings start with the Main module which is the entry point into the monitor. The monitor was written in C to maintain portability. The C programming language was commented as much as possible to enhance readability. The modules are presented in the approximate order in which they are used.

```
/* ----------------------- MONITOR.H   monitor header file ---------------------- */

#define     CONTINUE        1       /*  CONTINUATION FLAG     */
#define     HEX_ERR         2       /*  HEX CONVERSION ERROR  */
#define     MODIFY          3       /*  MEMORY MODIFY FLAG    */
#define     BS              0x08    /*  ASCII CODE FOR BACKSPACE */
#define     CR              0x0D    /*  ACCII CODE FOR RETURN  */
#define     NULL            0x00    /*  ASCII CODE FOR NULL    */
#define     SPACE           0x20    /*  ASCII CODE FOR SPACE   */

#ifdef MAINMON

char  BUFFIN[255];      /* 255 char input buffer   */
int   COUNT;
char  MONSTAT[3];       /*  RESERVE 3 BYTES FOR  STATUS */
long  START_ADDRESS
long  END_ADDRESS;      /*  RESERVE LONG WORDS FOR ADDRESSES  */
long  DATA;             /*  DATA FOR MEM MODIFY   */

/* messages */
char  *BKPTMSG = "BREAKPOINT TRAP AT ";
char  *ERRMSG = "ERROR RE-ENTER";
char  *HEXMSG = "HEX CONVERSION ERROR...RE-ENTER";
char  *ILLMSG = "ILLEGAL INSTRUCTION TRAP";
char  *MONMSG = "68000 MONITOR V0.0\rWRITTEN IN C BY BOB
                  VOIGT\r";
char  *REGERR = "REGISTER CONTENTS ERROR RE-ENTER";

char  *commands[ ] =            /* command string */
{"BKPT","RCHG","NOBK","REG1","QUIT","NL",
   "BK","GO","MD","MM"};

#else

extern      char  BUFFIN[ ],MONSTAT[ ];
extern      int   COUNT;
extern      long  BKPTAB[ ],START_ADDRESS,END_ADDRESS;
extern      long  DATA;
extern      char  *BKPTMSG,*ERRMSG,*HEXMSG;
extern      char   *ILLMSG,*MONMSG;
extern      char  *REGERR,*REGMSG[ ],*commands[];
#endif
```

```
/* ---------------------------------- MAIN ---------------------------------- */
/*    MAIN IS THE ENTRY POINT INTO THE MONITOR.       */

#define  MAINMON
#include  "monitor.h"     /* include header file */

monitor()
{
for(i=0; i<8;i++)
  MONSTAT[i] = 0;                  /*  Clear Monitor Status  */

 MESSAGE(MONMSG);          /* Output opening message */

while(TRUE)
{
 printf("\r$ ");              /* Print Prompt  "$"  */
 gets(BUFFIN);              /*  Enter monitor   */
 putchar(CR);
 CMD_DECODE();          /*  Decode input     */
 }

}       /* end main */

/* ---------------------- CMD_DECODE -------------------- */
/* THIS PROGRAM DECODES COMMANDS        */
/*  FROM THE COMMAND LINE                    */

CMD_DECODE()
{
char  match;

for(i=0;i<4;i++)              /* Convert all input commands to upper case */
  BUFFIN[i] = toupper(BUFFIN[i]);

for(i=0;i<5;i++)              /* look for 4 letter commands first */
{
match = strncmp(BUFFIN,commands[i],4);
         /* C function comparing first 4 letters of command string */
  if(match == 0)
    {
     switch(i)          /* if there's a match go to that routine */
     {
     case 0:            /* case numbers are indices of command string */
     BKPT();
     break;
```

```
      case 1:
       REGCHANG();
      break;

      case 2:
      NO_BKPT();
      break;

      case 3:
      REG1();
      break;

      case 4:
      ExitToShell();   /* QUIT */
      break;

     }                 /* end switch */
    break;
   }                   /* end if strncmp */
  }                    /* end for */

if(match != 0)        /* if no match in 4 letters */
{
for(i=5;i<10;i++)     /* try two letter commands */
{
match = strncmp(BUFFIN,commands[i],2);

 if(match == 0)
  {
    switch(i)
    {
    case 5:
    NL();
    break;

    case 6:
    BKPT_LIST();
    break;

    case 7:
    GO();
    break;

    case 8:
     MEM_DISPLAY();
    break;
```

```
    Start_ADD[i] = BUFFIN[j];          /* get string representation */
    i++;
    j++;
}

sscanf(Start_ADD,"%X",&START_ADDRESS);   /* convert string to hex  */

if(BUFFIN[j] == NULL)                  /* exit with return address = 0 */
  return;

i = 0;
j++;

 while(BUFFIN[j] != NULL)              /* string is terminated by a NULL */
 {
   if(!ishex(BUFFIN[j]))               /* check for proper hex format */
   {
     MESSAGE(HEXMSG);                  /* not proper format */
     MONSTAT[HEX_ERR] = TRUE;          /* exit with error message */
     return;
    }
    End_ADD[i] = BUFFIN[j];            /* get string representation of data */
    i++;
    j++;
 }

  if(MONSTAT[MODIFY])                  /* if modify  - convert to data */
     sscanf(End_ADD,"%X",&DATA);
   else                                /* else convert end address */
     sscanf(End_ADD,"%X",&END_ADDRESS);

}       /* end get_addr */


ishex(c)       /* routine to check if 0 < c < F  - a hex number */
char   c;
{
if((c >= 'a' && c<= 'f') || (c >= 'A' && c<= 'F') || isdigit(c))
 return(TRUE);
else
 return(FALSE);    /* returns a boolean true or false */
}

/* ---------------  MEM_MODIFY --------------      */
/*   THIS PROGRAM MODIFIES THE CONTENTS OF    */
/*       THE SPECIFIED MEMORY LOCATIONS           */
```

```
MEM_MODIFY()
{
MONSTAT[MODIFY]  = TRUE;        /* Set modify flag     */
MEM_DISPLAY();                  /* Display memory  */
MONSTAT[MODIFY]  = FALSE;       /* Clear modify flag */
}

/* -------------- MEM_DISPLAY --------------    */
/*  THIS PROGRAM DISPLAYS THE CONTENTS OF    */
/*       THE SPECIFIED MEMORY LOCATIONS          */

MEM_DISPLAY()
{
unsigned char     *byte,*databyte,*address,c;
long  *temp;

MONSTAT[HEX_ERR] = FALSE;       /* Clear errors */
GET_ADDR();                     /* Convert address to hex */

if(MONSTAT[HEX_ERR])            /* Was there a hex error ? */
   return;                      /*  yes, so exit  */

 /* make starting address a pointer to a byte */
byte = (unsigned char *)START_ADDRESS;

if (END_ADDRESS == 0)          /* if no end address then just list one row */
  END_ADDRESS = START_ADDRESS;

if(MONSTAT[MODIFY])            /* if modifying data */
{
databyte = (unsigned char *)&DATA;          /* data */
address = byte;                             /* use dummy variables */
for (i=0;i<4;i++)
 {
 if((*databyte && 0xFF) != 0)
   *address++ = *databyte++;
  else
   *databyte++;
  }
}

/* display address with changes  */

for(j=START_ADDRESS; j<= END_ADDRESS; j+=0x10)
{
 if(KeyReady())                 /* MAC specific routines  */
  {
```

```
    c = inKey();                    /* a backspace can stop the display */
    if(c == BS)
      return;
  }

  printf("\r%-8X    ",j);      /* print start address for the row */

    for(i=0;i<8;i++)            /* print 8 groups of 4 bytes with spaces between */
    {
     if(*byte == 0)             /* if whole byte is zero - print zeros */
     {
       printf("00");    /* need to fill in zeros with real zeros on screen  */
       *byte++;         /* get next byte       */
     }
    else
     {
     if((*byte & 0x00F0) == 0)      /* only upper nibble is a zero */
          printf("0");
       printf("%1X",*byte++);       /* print contents of byte    */
     }

    if(*byte == 0)      /* if whole byte is zero - print zeros */
     {
       printf("00 ");   /* these zeros have a space after them - formatted */
       *byte++;         /* get next byte       */
     }
    else
     {
     if((*byte & 0x00F0) == 0)      /* if upper nibble is zero again */
          printf("0");
        printf("%1X ",*byte++);  /* print byte with a space after it */
     }
    }
  }
}               /* end MEM_DISPLAY  */

/*------------------------------ OUTPUT_BYTE ------------------------- */
/*   THIS PROCEDURE CONVERTS A BYTE INTO 2 ASCII  */
 /*   CHARACTERS  AND SENDS THE CHARACTERS TO     */
/*     THE CRT DISPLAY                                                 */

OUTPUT_BYTE(byte)
char   byte
{
printf("%2X",byte);                 /* convert and print */
}
```

```
/*------------------------------------ MESSAGE ------------------------------------ */
/*  THIS PROCEDURE OUTPUTS MESSAGES TO THE CRT SCREEN */
MESSAGE(msgout)
char  *msgout;
{
printf("%s",msgout);              /* output message */
if(msgout == ERRMSG)              /* if there was a problem with the input */
 {
 printf("\rCOMMANDS ARE:\r");  /* remind  user of proper commands */
  for(i=0;i<10;i++)
      printf("%s ",commands[i]);
  }
}

/* ---------------------------------- STUBS ---------------------------- */
/*   THIS FILE CONTAINS PROGRAMMING STUBS  */

BKPT_LIST()
{
printf("in BKPT_LIST\r");
}

NO_BKPT()
{
printf("in NO_BKPT\r");
}

GO()
{
printf("in GO\r");
}


REGCHANG()
{
printf("in REGCHANG\r");
}

REG1()
{
printf("in REG1\r");
}

BKPT()
{
printf("in BKPT\r");
}
```

APPENDIX C

THE HARDWARE ENVIRONMENT

The operating system design was implemented on an Apple Macintosh personal computer. The operating system ran on the Apple Macintosh computer essentially as an application under the Macintosh's operating system. It did not take full control of the system, however it did modify the Macintosh's vector tables.

The Macintosh is not a DOS based computer. The Macintosh operating system is ROM based and the operating system kernel routines are in ROM. It contains a unique input/output setup and is by no means a development station for an imbedded MC68000 system.

Figure C.1 is a block diagram of the main logic board of the Macintosh. The Macintosh uses a great deal of Programmable Array Logic (PALS). The PALS, which are proprietary, may have a great deal to do with the operating speed of the Macintosh and contribute to the Macintosh not being an optimal system for a real time development station.

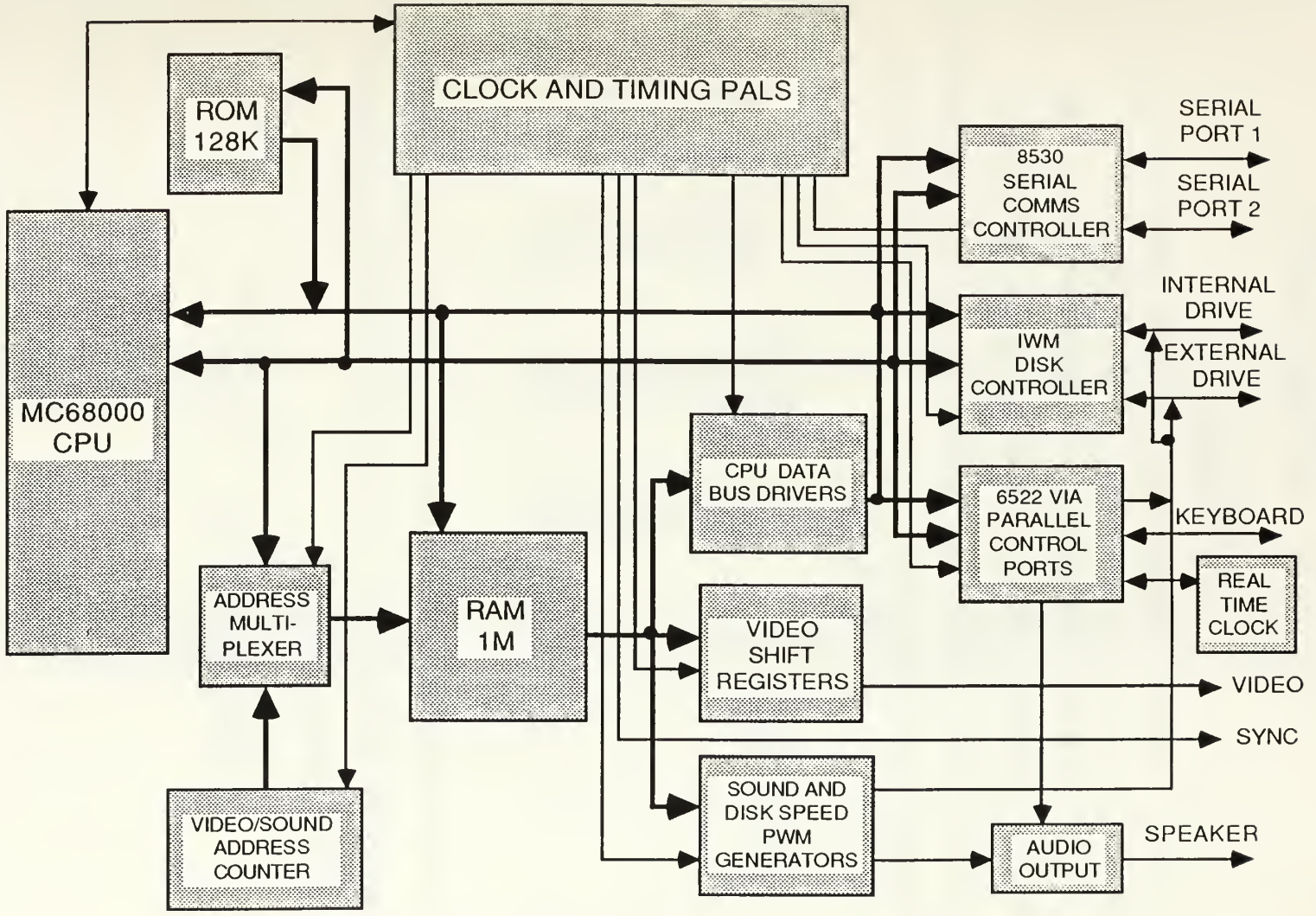


Figure C.1 Macintosh™ Logic Board Block Diagram

APPENDIX D

READY QUEUE PERFORMANCE DATA

The following data was recorded using the SY6522 timer. The execution times shown are the times necessary for the Scheduler to schedule eight jobs per minor cycle until the queue is full. The maximum jobs the queue could hold was fifty. The time was measured in E-Clock cycles, the E-Clock runs at 0.9792 MHz.

Number of Times Schedule Called	E-Clocks Run 1	Run 2	Run 3
1	2252	2241	2255
2	2392	2565	2392
3	2432	2389	2435
4	2599	2644	2556
5	2447	2675	2436
6	2478	2603	2476
7	1001	1001	999

The following data measurements are for the Serve routine. The data was acquired by starting with a full queue and serving one element at a time until the queue was empty.

Number of Elements in Queue Before Serve	Time to Serve in ms
50	0.561
49	0.534
48	0.546
47	0.649
46	0.549
45	0.543
44	0.523
43	0.564
42	0.559

Number of Elements in Queue Before Serve	Time to Serve in ms
41	0.556
40	0.545
39	0.544
38	0.551
37	0.543
36	0.548
35	0.526
34	0.550
33	0.523
32	0.444
31	0.458
30	0.451
29	0.469
28	0.543
27	0.438
26	0.455
25	0.460
24	0.453
23	0.463
22	0.361
21	0.438
20	0.448
19	0.364
18	0.426
17	0.419
16	0.383
15	0.420
14	0.359
13	0.352
12	0.304
11	0.355
10	0.312
9	0.349
8	0.362
7	0.273
6	0.324
5	0.269
4	0.238
3	0.202
2	0.165
1	0.170

LIST OF REFERENCES

1. National Aeronautics and Space Administration Technical Memorandum 72860, Digital Fly-By-Wire FlightControl Validation Experience, by Kenneth J. Szalai, Calvin R. Jarvis, Vincent A. Megna, Larry D. Brock, and Robert N. O'Donnell, December 1978.

2. National Aeronautics and Space Administration CR-163117, Advanced Flight Control System Study, by G. L. Hartmann, J. E. Hall, Jr., E. R. Rang, H. P. Lee, R. W. Schulte, and W. K. Ng, November 1982.

3. Dijkstra E.W., "The structure of T.H.E Multiprogramming System," Communications of the ACM, Vol 11, No. 5, May 1968.

4. Deitel, Harvey M., Operating Systems, Addison-Wesley Publishing Company, Inc., 1984.

5. Stubbs, Daniel F. and Webre, Neil W., Data Structures with Abstract Data Types and Pascal, Brooks/Cole Publishing Company, 1985.

6. Kernighan, Grian W. and Ritchie, Dennis M., The C Programming Language Prentice-Hall, Inc., 1978.

7. M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual, fifth edition, Prentice-Hall, Inc., 1986.

BIBLIOGRAPHY

Charles Stark Draper Laboratory CSDL-P-1727, Fault Tolerant Processor Concepts and Operation, by Basil T. Smith, 1 May 1983.

Freedman, A. L. and Lees, R.A., Real-Time Computing Systems, Crane, Russak and Company, Inc., 1977.

Inside Macintosh, Volumes I,II,III, and IV, Addison-Wesley Publishing Company, Inc., 1985.

Knuth, Donald E., The Art of Computer Programming, Volume 3, Searching and Sorting, Addison-Wesley Publishing Company, Inc., 1973.

Liebowitz, Burt H. and Carson, John H., Multiple Processor Systems for Real-Time Applications, Prentice-Hall, Inc., 1985.

NCR Microelectronics, NCR Microelectronics Data Book, 1985.

Shaw, Alan C., The Logical Design of Operating Systems, Prentice-Hall, Inc., 1985.

Tsichritzis, Dionysios C. and Bernstein, Philip A., Operating Systems, Academic Press, Inc., 1974.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Department Chairman, Code 62 Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
4.	Professor Larry W. Abbott, Code 62 At Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
5.	Professor Sherif Michael, Code 62 Mi Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
6.	Commanding Officer Attn: LT Robert J. Voigt Naval Underwater Systems Center Naval Education and Training Center Newport, Rhode Island 02841-5000	1

Thesis
V855 Voigt
c.1 The design of a real time operating system for a fault tolerant micro-computer.